Understanding Non-Blocking Atomic Commitment

Özalp Babaoğlu Sam Toueg

Technical Report UBLCS-93-2

January 1993

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)

The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` in compressed PostScript format. Abstracts are available from the same host in the directory `/pub/TR/ABSTRACTS` in plain text format. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`.

UBLCS Technical Report Series

92-1 *Mapping Parallel Computations onto Distributed Systems in Paralex*, by Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli, January 1992.

92-2 *Parallel Scientific Computing in Distributed Systems: The Paralex Approach*, by L. Alvisi, A. Amoroso, Ö. Babaoğlu, A. Baronio, R. Davoli and L. A. Giachini, February 1992.

92-3 *Run-time Support for Dynamic Load Balancing and Debugging in Paralex*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, September 1992.

92-4 *Paralex: An Environment for Parallel Programming in Distributed Systems*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, October 1992.

93-1 *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanism*, by Ö. Babaoğlu and K. Marzullo, January 1993.

93-2 *Understanding Non-Blocking Atomic Commitment*, by Ö. Babaoğlu and S. Toueg, January 1993.

Understanding Non-Blocking Atomic Commitment

Özalp Babaoğlu[1] **Sam Toueg**[2]

Technical Report UBLCS-93-2

January 1993

Abstract

In distributed database systems, an atomic commitment protocol *ensures that transactions terminate consistently at all participating sites even in the presence of failures. An atomic commitment protocol is said to be* non-blocking *if it permits transaction termination to proceed at correct participants despite failures of others. Protocols that have this property are desirable since they limit the time intervals during which transactions may be holding valuable resources. In this paper, we show how non-blocking atomic commitment protocols can be obtained through slight modifications of the well-known Two-Phase Commit (2PC) protocol, which is known to be blocking. Our approach is modular in the sense that both the protocols and their proofs of correctness are obtained by plugging in the appropriate reliable broadcast algorithms as the basic communication primitives in the original 2PC protocol. The resulting protocols are not only conceptually simple, they are also efficient in terms of time and message complexity.*

[1] Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna Italy. This author was supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project Number 6360 (BROADCAST), the United States Office of Naval Research under contract N00014-91-J-1219, IBM Corporation, Hewlett-Packard of Italy and the Italian Ministry of University, Research and Technology.
[2] Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, New York 14583 USA. This author was supported in part by the National Science Foundation under Grant Number CCR-9102231, IBM Corporation (Endicott Programming Laboratory) and the Italian National Research Council (CNR-GNIM) through a visiting professor grant.

1 Introduction

There are two principal reasons for structuring a data management system as a distributed system rather than a centralized one. First, the data being managed may be inherently distributed, as in the customer accounts database of a bank with multiple branches. Second, the data may be distributed to achieve failure independence for increased availability, as in a replicated file system.

When transactions update data in a distributed system, partial failures can lead to inconsistent results. For instance, in the banking example above, a transaction to transfer money between two accounts at different branches may result in the credit operation without performing the corresponding debit. In the replicated file system, a write operation may cause two replicas of the same file to diverge. It is clear that termination of a transaction that updates distributed data has to be coordinated among its participants if data consistency is to be preserved even in the presence of failures. The coordination that is required is specified by the *atomic commitment problem* [14].

Among the solutions proposed for this problem, perhaps the best known is the Two-Phase Commit (2PC) protocol [11,21]. While 2PC indeed solves the atomic commitment problem, it may result in *blocking* executions where a correct participant is prevented from terminating the transaction due to inopportune failures in other parts of the system [3]. During these blocking periods, correct participants will also be prevented from relinquishing valuable system resources that they may have acquired for exclusive use on behalf of the transaction. Thus, it is desirable to devise *non-blocking* solutions to the atomic commitment problem that permit correct participants to proceed and terminate the transaction under as many failure scenarios as possible.

It is well known that distributed systems with unreliable communication do not admit non-blocking solutions to the atomic commitment problem [11,25,16]. If communication failures are excluded, non-blocking protocols do exist [25,6,19,7,13] and are typified by the Three-Phase Commit (3PC) protocol of Skeen [25]. These non-blocking protocols are not only inherently more costly (in time) than their blocking counterparts [7], they are also much more complex to program and understand. For example, to prevent blocking, correct participants may need to communicate with each other and consider a large number of possible system states in order to proceed with the correct decision towards termination. Furthermore, protocols such as 3PC invoke sub-protocols for electing a leader [9] and determining the last process to fail [26], which themselves are complex and costly.

In this paper, we develop a family of non-blocking protocols to solve the atomic commitment problem. All of our protocols share the basic structure of 2PC and differ only in the details of the communication primitive they use to broadcast certain messages. By exploiting the properties of these broadcast primitives, we are able to achieve non-blocking without adding any complexity beyond that of 2PC. We complete this “compositional methodology” of protocol design by demonstrating algorithms that achieve the properties of the hypothesized broadcast primitives. These algorithms turn out to be variants of *uniform reliable broadcast* [23,4]. The modular approach we advocate results in non-blocking atomic commitment protocols that are easy to prove and understand. This conceptual economy is obtained without any performance penalties — the best of our protocols is as efficient as 2PC. Furthermore, our solutions are complete in the sense that no additional sub-protocols are needed to put them to practice.

In the next two sections, we define the distributed system model and the context for distributed transaction execution. Within this environment, the requirement of global consistency despite failures is formally specified as the atomic commitment problem in Section 4. A generic protocol based on Two-Phase Commit for solving the atomic commitment problem is described in Section 5 and serves as the generator for our future protocols. The first of a series of broadcast primitives we consider is defined through a collection of properties in given Section 6. In the same section, we illustrate a simple algorithm that achieves the required properties. Plugging in this algorithm to the generic protocol of the previous section results in the classical implementation of 2PC, which is proven correct in Section 7. In Section 8, we consider the issue of blocking and refine the atomic commitment problem specification to include the non-blocking property. Section 9 contains the key result of the paper where we show that if the simple broadcast primitive specification of Section 6 is extended to include a *Uniform Agreement* property, then any algorithm that achieves this broadcast can be plugged into the generic protocol to obtain a non-blocking atomic commitment protocol. An algorithm that indeed achieves this broadcast is given in Section 10. The issue of recovery from failures is the subject of Section 11. The performance analysis carried out in Section 12 of our basic non-blocking protocol leads to the two new broadcast algorithms of Sections 13.1 and 13.2 that improve on the message complexity and time delay, respectively. Related work, communication failures and possible extensions of our results are discussed in Sections 14 and 15 before concluding the paper. Appendix A presents a further optimization of the basic non-blocking protocol through yet another broadcast primitive that exploits *a priori* knowledge about the latest time at which imminent broadcasts may begin. Appendix B contains simple adjustments that can be made to the timeout constants so that our protocols continue to work with rate-bounded, unsynchronized local clocks.

2 System Model

We follow closely the model and terminology used in [3]. The distributed system consists of a set of sites interconnected through a communication network. Sites support computation in the form of processes that communicate with each other by exchanging messages. We assume a *synchronous* model of computation in the sense that bounds exists (and are known) for both relative speeds of processes and message delays.

At any given time, a process may be either *operational* or *down*. While operational, it follows exactly the actions specified by the program it is executing. Failures may cause operational processes to go down, after which they take no actions at all. This operational-to-down transition due to failures is called a *crash*. It is also possible for a process that is down to become operational again after executing a *recovery protocol*. When a process crashes, all of its local state is lost except for what it wrote in *stable storage* [21]. During recovery, the only information available to a process is the contents of this stable storage. A process is *correct* if it has never crashed; otherwise it is *faulty*.[3]

While processes may crash, we assume that communication is reliable. Furthermore, each message is received within δ time units (as measured in real-time) after being sent. This parameter δ includes not only the time required to transport the message by the network, but also the delays incurred in processing it at the sending and receiving processes. For the sake of exposition, we initially assume that every process has a local clock that advances at the same rate as real-time. As discussed in Appendix B, our results can be easily extended to

[3] The periods of interest for these definitions are the duration of the atomic commitment protocol execution.

```
% Some participant (the invoker) executes:
    send [T_START: transaction, $\Delta_c$, participants] to all participants

% All participants (including the invoker) execute:
    upon (receipt of [T_START: transaction, $\Delta_c$, participants])
        $C_{know}$ := local_clock
        % Perform operations requested by transaction
        if (willing and able to make updates permanent) then
            vote := YES
        else vote := NO
        % Decide COMMIT or ABORT according to atomic commitment protocol
        atomic_commitment(transaction, participants)
```

Figure 1. Distributed Transaction Execution Schema

systems where local clocks are not perfect but their rate of drift from real-time is bounded. Each local clock is only used to measure time intervals. Thus, we do not need to assume that clocks are synchronized with each other [18].

Given the above model and the assumption that communication is failure free, timeouts can be used to detect process failures. In particular, if a process does not receive a response to a message within 2δ time units (as measured on its local clock) after sending it, it can conclude that the destination process is faulty (i.e., it has crashed at least once).

3 Distributed Transactions

Informally, a distributed transaction (henceforth called a "transaction") is the execution of a program accessing shared data at multiple sites [21]. The isolated execution of a transaction in the absence of failures is assumed to transform the data from one consistent state to another. Logical isolation in the presence of concurrent transactions is typically formalized as a *serializable execution* [24] and is achieved through a *concurrency control protocol* [2]. In this paper, we focus on *failure atomicity* — preserving data consistency in the presence of failures — which is orthogonal to serializability.

For each transaction, the set of processes that perform updates on its behalf are called *participants*. Each participant updates data that are local to it. To conclude the transaction, participants must coordinate their actions so that either all or none of the updates to the data are made permanent. We consider only the so-called centralized version of this coordination where one of the participants acts as the *coordinator* in order to orchestrate the actions. We assume that each transaction is assigned a unique global identifier. For sake of simplicity, we will consider only one transaction at a time and omit explicit transaction identifiers from our notation. Obviously, in a system with multiple concurrent transactions, all messages and variables will have to be tagged with identifiers so as to be able to distinguish between multiple instances.

Figure 1 illustrates the schema governing distributed transaction execution. It will serve as the context for specifying and solving the atomic commitment problem. The transaction begins at a single participant called the *invoker*. The invoker distributes the transaction to its participants by sending them T_START messages containing a description of the transaction operations and the full list of participants. As soon as a participant receives a T_START message (in line 2 of Figure 1) it is said to "know" about the transaction. The local time at which this event happens is recorded in the variable C_{know} for future use. The invoker computes an upper bound for the interval of time that may elapse from the instant any participant knows about the transaction to the time the coordinator (not necessarily the same participant as the invoker) actively begins concluding it. This interval, denoted Δ_c, is also included in the T_START message.

After a participant performs the operations requested by the transaction, it uses a variable $vote$ to indicate whether it can install the updates. A YES vote indicates that the local execution was successful and that the participant is willing and able to make the updates to the data permanent. In other words, the updates have been written to stable storage so that they can be installed as the new data values even if there are future failures. A NO vote indicates that for some reason (e.g., storage failure, deadlock, concurrency control conflict, etc.) the participant is unable to install the results of the transaction as the new permanent data values. Finally, participants engage in the coordination step to decide the outcome of the transaction by executing an atomic commitment protocol.

We are not interested in the details of how a participant is chosen to become the coordinator of a transaction. All we require is that each transaction is assigned a coordinator in a manner satisfying the following three axioms:

AX1: At most one participant will assume the role of coordinator.

AX2: If no failures occur, one participant will assume the role of coordinator.

AX3: There exists a constant Δ_c such that no participant assumes the role of coordinator more than Δ_c real-time units after the beginning of the transaction.

Axioms AX1 and AX2 are simply statements about the syntactic well-formedness of transactions — the program should guarantee that no more than one participant ever reaches the code for the coordinator and, in the absence of failures, indeed one participant should execute this code. Axiom AX3 allows us to bound the duration of a transaction even when its coordinator crashes before taking any steps.

At this point, we describe the programming notation used in this paper. As can be seen in Figure 1, we use a pseudo-Pascal syntax with the usual sequential control flow structures. We denote concurrent activities as tasks separated by "//" enclosed within **cobegin** and **coend**. Communication is accomplished through the **send** and **receive** statements by supplying the message and the destination/source process name. In our protocols, all messages carry type identifiers, written in SMALL-CAPS, within the message body. We use "**send** m **to** $\mathcal{G}$" as a shorthand for sending message m one at a time to each process that is a member of the set $\mathcal{G}$. Note that we make no assumptions about the indivisibility of this operation. In particular, the sender may crash after having sent to some but not all members of the destination set. The receiver of a message may synchronize its execution with the receipt of a message in one of two ways. The **wait-for** statement is used to block the receiver until the receipt of a particular message. If the message may arrive at unspecified times and should be received without blocking the receiver, then the **upon** statement is appropriate. Actually, both the **wait-for** and **upon** statements can be applied to arbitrary asynchronous events and not just to message receipts. When the specified event occurs, execution proceeds with the body of the respective statement. In case of a blocking wait, an optional timeout may be set to trigger at a particular (local) time using the **set-timeout-to** statement. The timeout value in effect is that set by the most recent **set-timeout-to** before the execution of a

wait-for statement. If the event being waited for does not occur by the specified time, then the **on-timeout** clause of the **wait-for** statement is executed rather than its body. The body and the timeout clause of **wait-for** are mutually exclusive.

4 The Atomic Commitment Problem

The *atomic commitment problem* is concerned with bringing a transaction to a globally consistent conclusion despite failures. For each participant, its goal is to select among two possible decision values — COMMIT and ABORT. Deciding COMMIT indicates that all participants will make the transaction's updates permanent, while deciding ABORT indicates that none will. The individual decisions taken are irreversible. A COMMIT decision is based on unanimity of YES votes among the participants.

We formalize these notions as a set of properties that, together, define the atomic commitment problem:

AC1: All participants that decide reach the same decision.
AC2: If any participant decides COMMIT, then all participants must have voted YES.
AC3: If all participants vote YES and no failures occur, then all participants decide COMMIT.
AC4: Each participant decides at most once (that is, a decision is irreversible).

A protocol that satisfies all four of the above properties is called an *atomic commitment protocol*.

5 A Generic Atomic Commitment Protocol

Figure 2 illustrates a generic atomic commitment protocol, called ACP, that has the same structure as 2PC. It is generic in the sense that the details of **broadcast** used by the coordinator to disseminate the decision have not been specified. We will use this protocol to obtain others (including 2PC) by plugging in appropriate instances of the broadcast primitive.

The protocol consists of two concurrent tasks, one executed only by the coordinator (task 1) and the other executed by all participants, including the coordinator (task 2). The coordinator starts out by collecting the votes of participants by sending them VOTE_REQUEST messages. When a participant receives such a message, it "votes" by sending the value of local variable $vote$ to the coordinator. Phase 1 ends when the coordinator has votes from all participants. If a YES vote was received from all participants, then the decision is COMMIT; otherwise it is ABORT. In Phase 2, the coordinator disseminates the decision to all participants. If a participant voted NO in Phase 1, it can unilaterally decide ABORT. Otherwise it has to wait for the decision to arrive from the coordinator. If no decision arrives at a participant from the coordinator, it engages in a termination protocol in an attempt to conclude the transaction with the help of others. If a participant has not received a VOTE_REQUEST by the appropriate time, it can safely assume that the coordinator has crashed and unilaterally decide ABORT. The choice of the timeout periods will be discussed when we prove the correctness of specific instances of this generic protocol.

6 A Simple Broadcast Primitive: SB

A key step in the generic protocol of Figure 2 is the dissemination of the decision value to all participants by the coordinator in Phase 2. We call the primitive to achieve this dissemination

```
procedure atomic_commitment(transaction, participants)
    cobegin
    % Task 1: Executed by the coordinator
        send [VOTE_REQUEST] to all participants                % Including the coordinator
        set-timeout-to local_clock + $2\delta$
        wait-for (receipt of [VOTE: vote] messages from all participants)
            if (all votes are YES) then
                broadcast (COMMIT, participants)
            else broadcast (ABORT, participants)
        on-timeout
            broadcast (ABORT, participants)

//

    % Task 2: Executed by all participants (including the coordinator)
        set-timeout-to $C_{know} + \Delta_c + \delta$
        wait-for (receipt of [VOTE_REQUEST] from coordinator)
            send [VOTE: vote] to coordinator
            if (vote = NO) then
                decide ABORT
            else
                set-timeout-to $C_{know} + \Delta_c + 2\delta + \Delta_b$
                wait-for (delivery of decision message)
                    if (decision message is ABORT) then
                        decide ABORT
                    else decide COMMIT
                on-timeout
                    decide according to termination_protocol()
        on-timeout
            decide ABORT
    coend
end
```

Figure 2. ACP: A Generic Atomic Commitment Protocol

```
procedure broadcast(m, $\mathcal{G}$)

% Broadcaster executes:
    send [DLV: m] to all processes in $\mathcal{G}$
    deliver m

% Process p $\neq$ broadcaster in $\mathcal{G}$ executes:
    upon (receipt of [DLV: m])
        deliver m
end
```

Figure 3. SB1: A Simple Broadcast Algorithm

a **broadcast** which has a corresponding action at the destination called **deliver**. It is clear that **broadcast** and **deliver** will be implemented using multiple **send** and **receive** operations that the network provides.

The simplest way for a process p to broadcast a message m to the members of a set $\mathcal{G}$ is for p to sequentially send m to each process in $\mathcal{G}$. When a process in $\mathcal{G}$ receives such a message, it just delivers it.

It is easy to see that this simple broadcast algorithm, called SB1 (Figure 3), satisfies the following properties (with $\Delta_b = \delta$):

B1 (Validity): If a correct process broadcasts a message m, then all correct processes in $\mathcal{G}$ eventually deliver m.

B2 (Integrity): For any message m, each process in $\mathcal{G}$ delivers m at most once, and only if some process actually broadcasts m.

B3 (Δ_b-Timeliness): There exists a known constant Δ_b such that if the broadcast of m is initiated at real-time t, no process in $\mathcal{G}$ delivers m after real-time $t + \Delta_b$.[4]

We assume that each broadcast message m is unique. This could be easily achieved by tagging messages with the name of the broadcaster and a sequence number.

Any broadcast primitive that satisfies the above three properties is called a *Simple Broadcast* (SB). The primitive allows any process to broadcast any message at any time. In other words, there is no *a priori* knowledge of the broadcast times or of the identity of the broadcasters. Note that SB is not reliable — if the broadcaster crashes in the middle of a Simple Broadcast, it is possible for some correct processes to deliver the broadcaster's message while other correct processes never do so.

7 The Two-Phase Commit Protocol: ACP-SB

Let us first consider an instantiation of the generic protocol ACP obtained by plugging in any SB algorithm (such as SB1) as the broadcast primitive in Figure 2. The resulting protocol

[4] Note that Integrity prevents even *faulty* processes from delivering a message more than once or "out of thin air". Similarly, Timeliness prevents *faulty* processes from delivering m after real-time $t + \Delta_b$. Since these two properties impose restrictions on message deliveries not only by correct processes, but also by faulty ones, they are called " *Uniform* Integrity and *Uniform* Δ_b-Timeliness" in the terminology of [15]. In what follows, we omit the qualifier "Uniform" for the sake of brevity.

is called ACP-SB and corresponds exactly to the classical 2PC protocol. Since this protocol forms the basis for all others to come, we give a detailed proof of its correctness. Large portions of this proof will remain valid also for other protocols we develop based on ACP. For the purposes of this proof, we assume that the termination protocol, which is invoked if a timeout occurs while waiting for the decision from the coordinator, simply "blocks" the participant. We do this without any loss of generality since the protocols developed later will be able to decide unilaterally without needing a termination protocol.

Theorem 1 *Protocol ACP-SB achieves properties AC1–AC4 of the atomic commitment problem.*

Proof: We prove the properties in the order AC2, AC3, AC4 and AC1.

AC2: If any participant decides COMMIT*, then all participants must have voted* YES.

Assume some participant decides COMMIT. This can only occur in line 20, and the participant must have delivered a COMMIT message in line 17. By the Integrity property of the broadcast, COMMIT was broadcast by some participant. This can only occur at line 5. Thus the coordinator must have received votes from all participants and all these votes were YES.

AC3: If all participants vote YES *and no failures occur, then all participants decide* COMMIT.

Suppose all participants vote YES and no failures occur. Let t_{start} be the real-time at which the transaction begins (this is the time at which some participant sends a T_START message). From AX2 and AX3, one participant assumes the role of coordinator by real-time $t_{start} + \Delta_c$. This coordinator immediately sends VOTE_REQUEST messages which arrive by $t_{start} + \Delta_c + \delta$. Note that in line 11 each participant waits for this VOTE_REQUEST, with a timeout set to trigger $\Delta_c + \delta$ time units after the real-time, t_{know}, at which it first learned about the transaction (in line 2 of Figure 1)[5]. In other words, this timeout is set to trigger at real-time $t_{know} + \Delta_c + \delta$. Since $t_{start} \leq t_{know}$, each participant receives the VOTE_REQUEST message it was waiting for, before the timeout is ever triggered. Thus, all participants send their YES votes to the coordinator. These votes arrive at the coordinator within 2δ time units of its sending the VOTE_REQUEST. Therefore, the timeout associated with the coordinator's wait for votes in line 3 never triggers. So, the coordinator receives YES votes from all participants and broadcasts a COMMIT message to all participants by real-time $t_{start} + \Delta_c + 2\delta$. Note that in line 17, all correct participants are waiting for the delivery of this decision message, with a timeout set to trigger at real-time $t_{know} + \Delta_c + 2\delta + \Delta_b$. By the Validity and Δ_b-Timeliness properties of the broadcast, every participant delivers the COMMIT message by real-time $t_{start} + \Delta_c + 2\delta + \Delta_b$, before this timeout is triggered. Thus all participants decide COMMIT.

AC4: Each participant decides at most once.

From the structure of protocol ACP-SB, each participant decides at most once while executing Task 2.

AC1: All participants that decide reach the same decision.

For contradiction, suppose participant p decides COMMIT and participant q decides ABORT. By AC4, $p \neq q$. Participant q can decide ABORT only in lines 14, 19 and 24. By the proof of AC2, since p decides COMMIT, the coordinator must have received votes from all participants, including q, and all these votes were YES. Since q sent a YES vote, it could not have decided ABORT in lines 14 or 24. So it must have decided ABORT in line 19, following the delivery of an ABORT message. By the Integrity property of the broadcast, some participant

[5] Note that the variable C_{know} records the local time at which this event occurs.

c' must have broadcast this message. From the protocol it is clear that c' assumed the role of coordinator. Since by the protocol, a coordinator may broadcast at most one decision message, c must be different from c'. This contradicts axiom AX1, stipulating that each transaction has at most one coordinator. □

8 The Non-Blocking Atomic Commitment Problem

Recall that in protocol ACP-SB, if a participant times out waiting for the decision from the coordinator, it invokes a termination protocol. Informally, this protocol will try to contact some other participant that has already decided or one that has not yet voted. If it succeeds, this will lead to a decision. There will, however, be failure scenarios for which no termination protocol can lead to a decision [11,25].

For example, consider a ACP-SB execution where the coordinator crashes during the broadcast of the decision (in Phase 2 of Task 1). Suppose that:

- all faulty participants deliver the decision and then crash, and
- all correct participants have previously voted YES (in Phase 1 of Task 1), and they do not deliver the decision.

If faulty participants do not recover, no termination protocol can lead correct participants to decide: Any such decision may contradict the decision made by a participant that crashed. We say that an atomic commitment protocol is *blocking* if it admits executions in which *correct* participants cannot decide. The scenario above shows that ACP-SB is blocking.

As we have argued in the Introduction, blocking atomic commitment protocols are undesirable since they result in poor system resource utilization. An atomic commitment protocol is said to be *non-blocking* if it satisfies the following property in addition to AC1–AC4:

AC5: Every correct participant that executes the atomic commitment protocol eventually decides.

Note that the non-blocking property of atomic commitment protocol is stated in terms of *correct* and not *operational* participants. This is because an operational participant may have crashed and then recovered, in which case, decision is to be achieved through the recovery protocol rather than the commitment protocol. Furthermore, the property requires only those participants that execute the atomic commitment protocol to eventually decide. From the distributed transaction execution schema of Figure 1, there may be others that do not execute the protocol because they do not know about the transaction. For them, we do not insist on a decision since they are not holding any resources on behalf of the transaction.

9 The Non-Blocking Atomic Commitment Protocol: ACP-UTRB

We now show that protocol ACP-SB can be made non-blocking by replacing SB with a stronger broadcast primitive. Recall that ACP-SB leads to blocking only if the coordinator crashes while broadcasting a decision and this decision is delivered only by participants that later crash. Thus blocking can occur because SB (the broadcast used to disseminate the decision) allows faulty processes to deliver a message that is never delivered by correct processes. This undesirable scenario is prevented by using *Uniform Timed Reliable Broadcast* (*UTRB*), a broadcast primitive that requires

B4 (Uniform Agreement): If any process (correct or not) in $\mathcal{G}$ delivers a message m, then all correct processes in $\mathcal{G}$ eventually deliver m

```
procedure atomic_commitment(transaction, participants)
    cobegin
    % Task 1: Executed by the coordinator
        send [VOTE_REQUEST] to all participants                  % Including the coordinator
        set-timeout-to local_clock + $2\delta$
        wait-for (receipt of [VOTE: vote] messages from all participants)
            if (all votes are YES) then
                broadcast (COMMIT, participants)                      % Using a UTRB
            else broadcast (ABORT, participants)                      % Using a UTRB
        on-timeout
            broadcast (ABORT, participants)                           % Using a UTRB

//

    % Task 2: Executed by all participants (including the coordinator)
        set-timeout-to $C_{know} + \Delta_c + \delta$
        wait-for (receipt of [VOTE_REQUEST] from coordinator)
            send [VOTE: vote] to coordinator
            if (vote = NO) then
                decide ABORT
            else
                set-timeout-to $C_{know} + \Delta_c + 2\delta + \Delta_b$
                wait-for (delivery of decision message)
                    if (decision message is ABORT) then
                        decide ABORT
                    else decide COMMIT
                on-timeout
                    decide ABORT                              % Replaces termination protocol
        on-timeout
            decide ABORT
    coend
end
```

Figure 4. ACP-UTRB: A Non-Blocking Atomic Commitment Protocol Based on UTRB

in addition to the Validity, Integrity, and Δ_b-Timeliness of SB.[6] Note that with respect to message delivery, property B4 requires agreement among all processes, and not just those that are correct. It is this *uniformity* aspect of agreement that makes blocking scenarios impossible.

Figure 4 illustrates ACP-UTRB, a non-blocking atomic commitment protocol based on UTRB. This non-blocking protocol is obtained from ACP as follows:

- The coordinator uses UTRB (rather than SB) to broadcast the decision in Lines 5, 6 and 8, and
- if a participant times out while waiting to deliver this decision, it simply decides ABORT (rather than invoking a termination protocol) in Line 22.

Removing the termination protocol, the only source of indefinite wait in ACP-SB, eliminates blocking.

Theorem 2 *ACP-UTRB achieves properties AC1–AC4 of the atomic commitment problem.*

Proof: The proofs of AC2, AC3 and AC4 remain exactly the same as with ACP-SB. This is because UTRB has the Validity, Integrity and Δ_b-Timeliness properties of SB, and the modifications made to ACP in obtaining ACP-UTRB do not affect these proofs. The proof of AC1, however, is modified as follows.

The proof is by contradiction. Suppose participant p decides COMMIT and participant q decides ABORT. By AC4, $p \neq q$. Participant q can decide ABORT only in lines 14, 19, 22 and 24. The proof that q cannot decide ABORT in lines 14, 19 and 24 is exactly as before. Suppose that q decides ABORT at line 22, that is after timing out while waiting for the delivery of the decision message. Note that this timeout occurs at real-time $t_{know} + \Delta_c + 2\delta + \Delta_b$. From the first part of the proof, a coordinator must have broadcast a COMMIT message which was delivered by p. By AX3 and the protocol, this broadcast occurred by real-time $t_{start} + \Delta_c + 2\delta$. Since p delivered COMMIT, by the Uniform Agreement property of the broadcast, q eventually delivers COMMIT as well. By the Δ_b-Timeliness property of the broadcast, q does so by real-time $t_{start} + \Delta_c + 2\delta + \Delta_b$. Since $t_{start} \leq t_{know}$, q must have delivered COMMIT before timing out. This, however, contradicts the semantics of the **wait-for** statement. □

We now prove that ACP-UTRB is indeed non-blocking by showing that it satisfies AC5.

Theorem 3 *Every correct participant that executes ACP-UTRB eventually decides.*

Proof: In ACP-SB, a correct participant could be prevented from reaching a decision only by executing the termination protocol of line 22. This is because each **wait-for** statement has an associated timeout clause that makes indefinite waiting elsewhere impossible. In ACP-UTRB, we substituted the termination protocol with a unilateral ABORT decision, thus eliminating the only potential source of blocking. So, every correct participant that executes ACP-UTRB eventually decides. □

[6] Thus, UTRB is a strengthening of SB.

```
procedure broadcast(m, $\mathcal{G}$)

% Broadcaster executes:
    send [DLV: m] to all processes in $\mathcal{G}$
    deliver m

% Process p $\neq$ broadcaster in $\mathcal{G}$ executes:
    upon (first receipt of [DLV: m])
        send [DLV: m] to all processes in $\mathcal{G}$
        deliver m
end
```

Figure 5. UTRB1: A Simple UTRB Algorithm

10 A Simple UTRB Algorithm

In Figure 5 we show UTRB1, a simple UTRB algorithm obtained from SB1 as follows. First, each process relays every message it receives to all others (so, if any correct process receives a message, then all correct processes also receive it, even if the broadcaster crashes). Second, a process does not deliver a message it has received until it has completed relaying it (thus, all correct processes receive and deliver the message even if the relayer subsequently crashes).

Theorem 4 *Algorithm UTRB1 achieves broadcast properties B1–B4.*

Proof: The proofs that UTRB1 satisfies B1 and B2 (Validity, Integrity) are trivial. We now prove B3 and B4.

B3 (Δ_b-Timeliness): Let F denote the maximum number of processes that may crash during the execution of the atomic commitment protocol. Suppose the broadcast of message m is initiated at real-time t_b and some process, say p, delivers m. We show that there exists a constant delay $\Delta_b = (F+1)\delta$ by which this delivery must occur. Let $p_1, p_2, \ldots, p_i$ be the sequence of processes that relayed m from the broadcaster on its way to p, where $p_1 = broadcaster$ and $p_i = p$. Since a process never sends the same message more than once, these processes are all distinct. Note that for all j, $1 \leq j \leq i$, message m traverses $j-1$ links before it is delivered by a process p_j. So p_j delivers m by real-time $t_b + (j-1)\delta$. There are two cases to consider:

1. $(i \leq F+1)$ Process $p = p_i$ delivers m by time $t_b + F\delta$, that is within Δ_b of the broadcast.
2. $(i > F+1)$ Consider processes $p_1, p_2, \ldots, p_{F+1}$. Since they are all distinct, one of them, say p_j for some $j \leq F+1$, must be correct. Note that p_j delivers and relays m by time $t_b + (j-1)\delta$. Since p_j is correct, p will receive m from p_j at most δ time units later, that is by time $t_b + j\delta$. Since $j \leq F+1$, p_j delivers m by time $t_b + (F+1)\delta$, that is within Δ_b of the broadcast.

B4 (Uniform Agreement): To show that UTRB1 satisfies B4, note that a process does not deliver a message unless it has previously relayed that message to all. So, if any process delivers a message, this message will eventually be received and delivered by all correct processes. □

```
procedure recovery_protocol(p)
% Executed by recovering participant p
    R := set of DT-log records regarding transaction
    case R of
        {}:                         skip
        {start}:                    decide ABORT
        {start,no}:                 decide ABORT
        {start,vote,decision}:      skip
        {start,yes}:
            while (undecided) do
                send [HELP, transaction] to all participants
                set-timeout-to $2\delta$
                wait-for receipt of [REPLY: transaction, reply] message
                    if (reply $\neq$?) then
                        decide reply
                    else
                        if (received ? replies from all participants) then
                            decide ABORT
                on-timeout
                    skip
            od
    esac
end
```

Figure 6. Recovery: The Recoverer's Algorithm

11 Recovery from Failures

To complete our discussion of non-blocking atomic commitment, we need to consider the possibility of a participant that was down becoming operational after being repaired. Such a participant returns to the operational state by executing a *recovery protocol*. This protocol first restores the participant's local state using a *distributed transaction log* (DT-log) that the participant maintains in stable storage. The protocol then tries to conclude all the transactions that were in progress at the participant at the time of the crash. Our recovery protocol and DT-log management scheme are very similar to those of 2PC with cooperative termination [3]. We include them here for completeness.

We consider the possibility of recovery by adding the following requirement to the specification of the atomic commit problem:

AC6: If all participants that know about the transaction remain operational long enough, then they all decide.

To facilitate recovery, participants write records of certain key actions in the DT-log. In particular, when a participant receives a T_START message, it writes a **start** record containing the transaction identifier and the list of participants in the DT-log. Before a participant sends a YES vote to the coordinator, it writes a **yes** record in the DT-log. If the vote is NO, a **no** record is written as the vote and an **abort** is written as the decision. These records can be written before or after sending the NO vote. When a COMMIT (or ABORT) decision is received from the coordinator, the participant writes a **commit** (or **abort**) record in the DT-log. Writing of

```
upon (receipt of [HELP, transaction] message from p)
    R := set of DT-log records regarding transaction
    case R of
        {}:                          decide ABORT; send [REPLY: transaction, ABORT] to p
        {start}:                     decide ABORT; send [REPLY: transaction, ABORT] to p
        {start,no}:                  send [REPLY: transaction, ABORT] to p
        {start,vote,decision}:       send [REPLY: transaction, decision] to p
        {start,yes}:                 send [REPLY: transaction, ?] to p
    esac
```

Figure 7. Recovery: The Responder's Algorithm

this decision record constitutes the act of "deciding." In the protocol descriptions, we use **vote** and **decision** to represent arbitrary vote and decision records, respectively.

The recovery protocol consists of two components — the actions performed by the recovering participant (Figure 6) and the actions performed by other participants in response to requests for help (Figure 7). For each transaction that was active at the time of the crash, the recovering participant first tries to decide unilaterally based on the DT-log. If it cannot, it send out requests for help from the other participants until it either receives a decision from some participant, or it receives "don't know" replies from all participants. When this protocol is used for recovery together with our non-blocking ACP-UTRB, we can prove the following property (which is stronger than AC6):

Theorem 5 *(AC6′) If a participant that knows about the transaction recovers, it will eventually decide provided that either (i) there was no total failure, or (ii) there was a total failure but all participants recover and stay operational long enough.*

Proof: Let p be the recovering participant. The only case where p cannot decide unilaterally is if crashed after having voted YES but before having decided. We show that in both scenarios of the theorem, p eventually decides.

Suppose participant p is recovering from a partial failure. In other words, there exists a participant q that has never crashed. There are two cases to consider:

1. (Participant q knows about the transaction) Since ACP-UTRB is non-blocking (i.e., satisfies AC5), q eventually decides. The recovering participant p succeed in contacting q for help and will decide accordingly.
2. (Participant q does not knows about the transaction) By the protocol, when asked by p, q will unilaterally decide ABORT and send it to p, forcing it to decide.

Now suppose p is recovering from a total failure. Note that p repeatedly sends help messages until either it receives a $reply \neq ?$ or it receives a reply from every participant: both cases allow p to decide. Since we assume that all participants eventually recover and remain operational for long enough to respond to p's request for help, one of these two conditions must eventually occur, and p eventually decides. □

Note that case in recovering from a total failure, waiting for all participants to recover is conservative in the sense that it may be possible to recover earlier. In particular, it is sufficient to wait until the set of recovered participants contains the last one to have crashed [3,26].

Since the recovery protocol may lead to a decision, we need to show that this decision is not in conflict with any decisions resulting from the ACP-UTRB itself.

Theorem 6 *The recovery protocol of Figures 6 and 7 is correct (i.e., it does not violate properties AC1–AC4 of the atomic commitment problem).*

Proof: Note that we need not consider property AC3 since there could not have been any recovery by the assumption of no failures. We prove the remaining properties in the order AC4, AC2 and AC1.

(AC4) From the structure of the protocol and the use of DT-log, it is clear that even with recovery, participants decide at most once. Thus, AC4 remains valid.

(AC2) For contradiction, assume that AC2 is violated. Let p be that first participant that decides COMMIT during recovery, violating AC2. From the recovery protocol, p must have received a reply from some participant q with value COMMIT. By definition of p, q could not have decided during recovery (it must have done so using ACP-UTRB). All decisions reached using ACP-UTRB satisfy AC2. A contradiction.

(AC1) There are two case to consider:

1. (Some participant p decides d during ACP-UTRB) Let q be the first participant that decides during recovery. For contradiction, assume q decides $\bar{d}$. We again consider two cases.
   (a) (q decides unilaterally) Since ABORT is the only possible unilateral decision in the recovery protocol, it must be that $\bar{d}$ = ABORT, and so d = COMMIT. Since p decided COMMIT and ACP-UTRB satisfies AC2, all participants, including q, must have voted YES. So when q recovers, its DT-log contains $\{$**start**,**yes**$\}$. Since q unilaterally decides ABORT, it must have received "?" replies from all processes, including one from p. This is impossible since p decided COMMIT (and it's reply to q would also be COMMIT).
   (b) (q decides based on a reply) Let r be the participant that replies with the decision $\bar{d}$ to q. By the definition of q, r could not have decided during recovery. It must have done so in ACP-UTRB. Thus, p and r decide different values in ACP-UTRB. This is a contradiction since ACP-UTRB satisfies AC1.
2. (No participant decides in ACP-UTRB) We can show that all participants must decide ABORT. For contradiction, let p be the first participant that decides COMMIT during recovery. From the protocol, p must have received a COMMIT reply from some q. Thus, q must have decided before p. This contradicts the definition of p. □

12 Performance of ACP-UTRB

Just as its development and proof of correctness, the performance of the non-blocking atomic commitment protocol ACP-UTRB can be analyzed in a modular fashion. Both the time delay and message complexity of ACP-UTRB can be expressed as the sum of the cost of ACP and the cost of the particular instance of UTRB used.

Let n denote the number of participants for the transaction. Recall that F is the maximum number of participants that may crash during the protocol execution. Let $T_{ACP-UTRB}$ and $M_{ACP-UTRB}$ denote the time delay and message complexity, respectively, of ACP-UTRB. From the structure of ACP-UTRB, it is clear that $T_{ACP-UTRB} = 2\delta + \Delta_b$ and $M_{ACP-UTRB} = 2n + \mu_b$, where Δ_b and μ_b denote the timeliness and message complexity, respectively, of the particular UTRB algorithm used. The additive terms 2δ and $2n$ are the time and message costs due to ACP.

We now consider the performance of ACP-UTRB when UTRB1 is used as the broadcast algorithm. In UTRB1, each process relays each message to all other participants, so the message complexity of UTRB1 is n^2. The proof of Theorem 4 shows that the timeliness of UTRB1 is $(F+1)\delta$. Thus, the performance of ACP-UTRB using UTRB1 is $T_{ACP-UTRB1} = (F+3)\delta$ and $M_{ACP-UTRB1} = 2n + n^2$.

13 Optimizations

As we saw in the previous section, the performance of ACP-UTRB depends on the performance of the particular implementation of UTRB that is used. The implementation that we gave so far, UTRB1, is very simple but requires a quadratic number of messages. In the next two sections, we present more efficient implementations of UTRB. We first describe UTRB2, an algorithm that requires only a linear number of messages. We then give UTRB3, a message-efficient algorithm that improves on the timeliness of UTRB2.

13.1 A Message-Efficient UTRB Algorithm

The message complexity of algorithm UTRB1 can be reduced from quadratic to linear using the idea of *rotating coordinators* [4]. Rather than having each process relay every message to all other processes under all circumstances, we arrange for a process to assume the role of the initial broadcaster only in case of failures. The resulting algorithm is called UTRB2 and is displayed in Figure 8. The algorithm relies on the FIFO property of the communication channels.

The algorithm uses three types of messages: MSG announces the initial message, DLV causes a delivery, and REQ is used to request help. The initial broadcaster constructs a list of processes called *cohorts* that will cooperate in performing the broadcast. The first process on this cohort list is the broadcaster itself. To tolerate the failure of up to F processes, the cohort list contains $F+1$ distinct process names. This list, along with the index of the current cohort is included in MSG and REQ messages.

13.1.1 Performance of Algorithm UTRB2

Recall that Δ_b, the timeliness of UTRB, is the maximum time that may elapse between the broadcast and delivery of a message. In other words, *if* a message is delivered, it is delivered within Δ_b time units after the broadcast (but it is possible for a message broadcast by a faulty process not to be delivered by any process). Let f denote the number of processes that *actually* crash just during the execution of the broadcast algorithm. Clearly $f \leq F$, since F denotes the *maximum* number of processes that may crash during the entire atomic commitment protocol execution. We now derive expressions for Δ_b, and the message complexity of UTRB2, as a function of f.

The broadcaster sends a MSG message immediately followed by a DLV message to all. We assume that τ time units elapse between these two **send to all** operations. This time

```
procedure broadcast(m, $\mathcal{G}$)

% Broadcaster executes:
    send [MSG: m, cohorts, 1] to all processes in $\mathcal{G}$
    send [DLV: m] to all processes in $\mathcal{G}$

% Process p in $\mathcal{G}$ executes:
    upon (first receipt of [MSG: m, cohorts, index])
        i := index
        first_timeout := local_clock + $(\delta + \tau)$
        for k := $0, 1, \ldots$ do
            set-timeout-to first_timeout + k$(2\delta + \tau)$
            wait-for (receipt of [DLV: m])
                deliver m
                exit loop
            on-timeout
                if (i $<$ F+ 1) then
                    i := i + 1
                    send [REQ: m, cohorts, i] to cohorts[i]
                else exit loop                          % More than F cohorts have failed
        od

    upon (first receipt of [REQ: m, cohorts, index])
        send [MSG: m, cohorts, index] to all processes in $\mathcal{G}$
        send [DLV: m] to all processes in $\mathcal{G}$
end
```

Figure 8. UTRB2: A Message-Efficient UTRB Algorithm

accounts only for the processing delays and does not include network transport delays. [7] If there are no failures ($f = 0$), each process will receive this DLV message within $\delta + \tau$ time units from the time MSG was broadcast. This scenario results in a total of $2n$ messages.

Now consider the case $f = 1$ and the broadcaster is faulty. The worst-case message delay occurs if at least one process receives the broadcaster's MSG message but not all receive the DLV. The initial MSG message could take up to δ time units to arrive. Those processes that receive MSG but do not receive DLV will wait an additional $\delta + \tau$ time units from the time they received MSG before timing out and requesting help from the next cohort. The cohort receives this request at most δ time units later. This cohort has to be correct (since $f = 1$) and sends MSG followed by DLV at most τ time units after it received the request. This DLV is received at most δ time units later. Thus, the maximum total elapsed time before delivery becomes $\delta + (\delta + \tau) + \delta + \tau + \delta = 4\delta + 2\tau$. As for messages, note that only those processes that did not receive DLV send REQ messages to the next cohort. Thus, the number of MSG and DLV messages sent by the broadcaster and the REQ messages sent by processes to the next cohort sum to $2n$. The cohort behaves just like the original broadcaster and sends $2n$ additional messages, resulting in $4n$ total messages.

Number of Faulty Processes	*Timeliness* (Δ_b)	*Messages*
$f = 0$	$\delta + \tau$	$2n$
$1 \leq f \leq F$	$(f + 1)(2\delta + \tau)$	$(f + 1)2n$

Table 1. Performance of Algorithm UTRB2

In general, worst-case performance results when each additional failure is that of a different cohort. The loop with a $2\delta + \tau$ timeout period is repeated until either a DLV message arrives (and causes delivery), or there are no more cohorts to ask for help (i.e., more than F processes crash). Thus, each new failure beyond the first results in $2\delta + \tau$ additional time units and $2n$ additional messages. These results are summarized in Table 1.

13.1.2 Correctness of Algorithm UTRB2

We now prove the correctness of algorithm UTRB2.

Theorem 7 *Algorithm UTRB2 of Figure 8 satisfies the UTRB properties.*

Proof: We show that UTRB2 satisfies the Validity, Integrity, Δ_b-Timeliness and Uniform Agreement.

Validity: Assume that the broadcaster is correct. It sends [MSG: m, $cohorts$, 1] at real time t_b to all, and the [DLV: m] message by time $t_b + \tau$ to all. Since the channels are FIFO, all correct processes will first receive [MSG: m, $cohorts$, 1] (in line 3) and then receive [DLV: m] (in line 8) at most $\delta + \tau$ time units later. Thus every correct process delivers m without timing out on the wait of line 8.

Integrity: Clear from the structure of the algorithm.

[7] In systems where the communication subsystem buffers messages such that send operations do not block a process, τ should be negligible compared to δ.

Δ_b-Timeliness: Recall that at most F processes may crash during the atomic commitment protocol execution. In the worst case, $f = F$ processes actually crash during the execution of algorithm UTRB2. From Table 1, we see that if F processes crash then any message broadcast at time t cannot be delivered after time $t + \Delta_b$, where $\Delta_b = (F+1)(2\delta+\tau)$.

Uniform Agreement: It is trivially satisfied if no process ever delivers a message. So, suppose some process delivers a message m. By Integrity, m was broadcast by some process. Moreover, no process can deliver a message other than m. Thus, we only need to show that all correct processes eventually deliver some message.

For contradiction, let q be a correct process that never delivers a message. Since some process delivered a message, at least one of the cohorts sent a DLV message. This cohort must have sent a MSG message to all before sending this DLV. Thus, q enters and exits the loop that starts in line 6. Let t_1 and t_2 be the real-times at which q enters and exits, respectively, this loop. Note that t_1 is the time that q received its first MSG message, and that, since q waits $\delta + \tau$ units on its first iteration of the loop, $t_1 + (\delta + \tau) \leq t_2$. Since q never delivers a message, it did not receive any DLV message during the interval of time $[t_1, t_2]$.

Claim 1 *For all j, $3 \leq j \leq F+1$, if a* REQ *message is sent to c_j, then a* REQ *message was also sent to c_{j-1} at least δ time units earlier.*

Proof: Let s be the first process that sent REQ message to c_j. From the algorithm, it is easy to see that one of these two cases holds:

1. Process s itself sent a REQ message to c_{j-1}, and it did so at least δ time units earlier.
2. The $index$ field of the first MSG message that s received is $j-1$, and it came from c_{j-1}. From the receipt of that message, s waits for $\delta + \tau$ before sending the REQ message to c_j. Note that c_{j-1} must have received some REQ message before sending the MSG message to s. Thus, the REQ to c_{j-1} was sent at least $\delta + \tau$ before the REQ to c_j.

So the Claim holds in both cases. □

Claim 2 *For all i, j, $1 \leq i < j \leq F+1$, if both c_i and c_j send* MSG *messages, then c_i does so before (or at the same time as) c_j.*

Proof: For $i = 1$, c_i is the broadcaster and its first step is to send MSG messages to all. This clearly occurs before the other cohorts send any message. For $i \geq 2$, by Claim 1, the first REQ to c_i was sent at least δ earlier than the first REQ to c_j. Even if the REQ to c_i takes δ time units (the maximum possible), and the one to c_j arrives immediately, c_i must receive its REQ before (or at the same time as) c_j. Note that it is the receipt of the first REQ message that triggers the sending of a MSG. □

Claim 3 *A* REQ *message must have been sent to $c_2, \ldots, c_{F+1}$.*

Proof: Since q enters and exits the loop that starts in line 6, there are two possible cases:

1. Process q sent a REQ to c_{F+1}.
2. The $index$ field of the first MSG message that q received is $F+1$, and it came from c_{F+1}. Note that some REQ message must have been previously sent to c_{F+1}.

In both cases, by repeated application of Claim 1, we conclude that a REQ message was also sent to $c_2, \ldots, c_F$. □

Claim 4 *At least one* correct *cohort in* $\{c_1, c_2, \ldots, c_{F+1}\}$ *sent a* MSG *followed by a* DLV *to all (including* q*).*

Proof: Obvious from Claim 3 and the fact that at most F processes may be faulty. □

Let c_i be the correct cohort with the smallest index that satisfies Claim 4. We now show that the DLV sent by c_i is received by q during the interval $[t_1, t_2]$, contradicting our earlier remark. This is done by the next two Claims.

Claim 5 *The* DLV *sent by* c_i *arrives at* q *after* t_1*.*

Proof: Cohort c_i sent MSG before sending DLV. By the FIFO property, q receives MSG before receiving DLV. The result follows from the definition of t_1. □

Claim 6 *The* DLV *sent by* c_i *arrives at* q *before* t_2*.*

Proof: Consider the MSG that causes q to enter the loop (this occurs at time t_1). Let j be the value of the $index$ field of that MSG. This message is from cohort c_j. There are three possible case:

1. ($1 \leq j < i \leq F+1$) In this case, we can see from the algorithm that q eventually sends a REQ to c_i. Let t_{req} be the time it does so. Note that after t_{req}, process q waits $2\delta + \tau$ to receive a DLV. So, $t_{req} + 2\delta + \tau \leq t_2$. Note that c_i receives the REQ from q by time $t_{req} + \delta$, and being correct it sends a DLV by time $t_{req} + \delta + \tau$. This DLV arrives at q by time $t_{req} + 2\delta + \tau$, that is, before t_2.
2. ($1 \leq j = i \leq F+1$) In this case, the MSG that causes q to enter the loop at time t_1 is from c_i itself. Since c_i is correct, its DLV message (sent at most τ after MSG) is received by q by time $t_1 + \delta + \tau \leq t_2$.
3. ($1 \leq i < j \leq F+1$) Let s^i_{msg}, s^j_{msg} be the times that c_i and c_j sent their MSG message, respectively. From Claim 2, $s^i_{msg} \leq s^j_{msg}$. Note that c_i sends DLV by time $s^i_{msg} + \tau$, and this message arrives at q by time $r_{dlv} = s^i_{msg} + \tau + \delta$. Since $s^j_{msg} \leq t_1$, $r_{dlv} \leq t_1 + \tau + \delta \leq t_2$.

Thus, in all possible cases, the Claim holds. □

13.2 Reducing Time Delay: Algorithm UTRB3

We can modify UTRB2 to improve timeliness while maintaining a linear number of messages. The basic idea is to overlap sending of REQ messages to the next cohort with the (possible) arrival of DLV messages from the previous cohort. By being pessimistic and asking for help before the full round-trip message delay ($2\delta + \tau$), a process can reduce the time to deliver a message. The resulting algorithm of Figure 9, called UTRB3, is essentially UTRB2 with the timeout period for the request loop reduced from $2\delta + \tau$ to δ. Note that this modification may result in increased message traffic if the pessimism is unwarranted in the sense that the previous cohort was correct but REQ messages were sent before waiting long enough for the arrival of its DLV messages.

13.2.1 Performance of Algorithm UTRB3

In case of no failures, UTRB3 behaves exactly like UTRB2 and has the same timeliness and message complexity. In the general case, we can show that worst-case delay results when

```
procedure broadcast(m, $\mathcal{G}$)

% Broadcaster executes:
    send [MSG: m, cohorts, 1] to all processes in $\mathcal{G}$
    send [DLV: m] to all processes in $\mathcal{G}$

% Process p in $\mathcal{G}$ executes:
    upon (first receipt of [MSG: m, cohorts, index])
        i := index
        first_timeout := local_clock + $(\delta + \tau)$
        for k := $0, 1, \ldots$ do
            set-timeout-to first_timeout + k$\delta$
            wait-for (receipt of [DLV: m])
                deliver m
                exit loop
            on-timeout
                if (i $<$ F+ 1) then
                    i := i + 1
                    send [REQ: m, cohorts, i] to cohorts[i]
                else
                    if (k $\neq$ 0) then
                        set-timeout-to first_timeout + k$\delta + \delta + \tau$
                        wait-for (receipt of [DLV: m])
                            deliver m
                        on-timeout
                            skip                    % More than F cohorts have failed
                    exit loop
        od

    upon (first receipt of [REQ: m, cohorts, index])
        send [MSG: m, cohorts, index] to all processes in $\mathcal{G}$
        send [DLV: m] to all processes in $\mathcal{G}$
end
```

Figure 9. UTRB3: A Message- and Time-Efficient UTRB Algorithm

the message transits a chain of f faulty cohorts starting with the broadcaster before arriving at a correct one.[8] Thus, we need to quantify the maximum time that can elapse before some process sends a REQ message to the first correct cohort. Given the structure of the algorithm, a correct process requests help from a new cohort every δ time units after it fails to receive the DLV message from the broadcaster. At most $2\delta + \tau$ time units elapse until the first request for help and from then on a new cohort is involved every δ time units. Given that $f - 1$ faulty cohorts are woken up (since the broadcaster itself must be also faulty), we have $2\delta + \tau + (f - 1)\delta$ time units until the the first correct cohort is contacted for help. It is clear that within $2\delta + \tau$ time units after a process sends a REQ message to the first correct cohort, all processes deliver. Summing this final delay to the previous expression, we obtain $3\delta + 2\tau + f\delta$ as the timeliness parameter.

Consider the message complexity for $f = 1$. As before, the MSG and DLV messages sent by the (faulty) broadcaster plus the REQ messages sent to the first cohort sum to $2n$. From the instant the first REQ message is sent to the time when the corresponding DLV message arrives ($2\delta + \tau$ time units later), up to two additional cohorts can be woken up (resulting in at most n REQ messages each). Thus, up to three cohorts may be active. Each of these three (correct) cohorts will respond by sending $2n$ messages. Summing up, we have $10n$ total messages. Generalizing this analysis, we obtain $8n + 2fn$ as the message complexity of UTRB3 for the case $1 \leq f \leq F$. The performance of UTRB3 is summarized in Table 2.

Number of Faulty Processes	*Timeliness* (Δ_b)	*Messages*
$f = 0$	$\delta + \tau$	$2n$
$1 \leq f \leq F$	$3\delta + 2\tau + f\delta$	$8n + 2fn$

Table 2. Performance of Algorithm UTRB3

13.2.2 Correctness of Algorithm UTRB3

We now prove the correctness of algorithm UTRB3.

Theorem 8 *Algorithm UTRB3 of Figure 9 satisfies the UTRB properties.*

Proof: The proofs of Validity and Integrity are exactly as those for algorithm UTRB2. The proof of Δ_b-Timeliness is obtained by substituting F for f in the delay analysis of the previous section, resulting in $\Delta_b = 3\delta + 2\tau + F\delta$.

The proof of Uniform Agreement is the same as that for UTRB2, except for the first case of Claim 6, which is now as follows.

1. ($1 \leq j < i \leq F + 1$) In this case, we can see from the algorithm that q eventually sends a REQ to c_i. Let t_{req} be the time at which it does so. Note that after t_{req}, process q first waits δ to receive a DLV. If $i = F + 1$, then q waits an additional $\delta + \tau$ units of time (at line 18) before exiting the loop. If $i < F + 1$, then q eventually sends a REQ to c_{F+1} and waits $\delta + (\delta + \tau)$ before exiting the loop. Thus, in both cases, $t_{req} + 2\delta + \tau \leq t_2$.

[8] Actually there is another scenario leading to the same worst-case delay. Here, the message transits an alternating chain of cohorts and participants, all faulty except the final cohort.

Note that c_i receives the REQ from q by time $t_{req} + \delta$, and being correct it sends a DLV by time $t_{req} + \delta + \tau$. This DLV arrives at q by time $t_{req} + 2\delta + \tau$, that is, before t_2. □

14 Related Work

The role of reliable broadcast (or other formulations including Byzantine Agreement [20]) in distributed database transaction processing has been the subject of numerous works [19, 22,10,5,12,14]. Most of these studies have tried to relate the atomic commitment problem of transaction processing to various formulations of the Byzantine Agreement (BA) problem. In others, BA has been proposed as a way to relax the synchronous system assumptions or permit failures that are more general than the crash model. For instance, in [5], the transaction commitment problem is formulated in an "almost asynchronous" system and solved using a randomized BA protocol [1]. It is well known that in such a system, no deterministic solution to the atomic commitment problem exists that can tolerate even a single crash failure [8]. In [10] BA is used to cope with data storage nodes that may fail in an arbitrary and malicious manner.

Perhaps the work that is most similar in spirit is [22] where BA is is used to replace the second phase of 2PC. The motivation for the work, however, is to reduce recovery time at the cost of increased message traffic and longer delays for deciding. Moreover, no formal specifications are given for either the atomic commitment problem or the BA used in the protocol.

15 Discussion

As stated earlier, distributed systems with unreliable communication do not admit non-blocking solutions to the atomic commitment problem. Our protocols are no exception — if communication is not reliable, they have to block in order not to violate property AC1. The same blocking scenario of Section 8 where participants are partitioned into two groups may result as a consequence of communication failures even if no participant crashes. In this case, undecided correct participants cannot proceed because the other group is disconnected due to communication failures rather than its participants being down. In terms of our modular construction, the possibility of blocking can be explained by noting that communication failures render the Uniform Agreement property (B4) of UTRB broadcast impossible to achieve. Given the impossibility result, the best we can hope for is to extend our protocols such that participants in a majority partition are able to proceed towards a decision while others remain blocked [17].

16 Conclusions

We have described a solution to the non-blocking atomic commitment problem that is as easy to understand and prove as the well-known Two-Phase Commit protocol, a protocol that may lead to blocking [11,21]. Furthermore, our solution is *complete*: it does not require any additional protocols that other solutions typically require (e.g., the Three-Phase Commit Protocol requires a fault-tolerant leader election protocol [25]).

This solution was derived by focusing on the *properties* of the broadcast primitive used to disseminate the decision values. Indeed, it was obtained just by strengthening the broadcast used in the second phase of the Two-Phase Commit protocol.

Our non-blocking atomic commitment protocol, ACP-UTRB, was given modularly: a generic ACP protocol that relies on the properties of the UTRB broadcast. To demonstrate the practicality of ACP-UTRB, we have given several implementations of UTRB, each improving some performance measure. The performance of the resulting ACP-UTRB implementations are comparable to 2PC, yet they are non-blocking.

Appendix

A Eliminating Decisions Based on Timeouts

Recall that the time performance of ACP-UTRB is given by the expression $2\delta + \Delta_b$ where Δ_b is the timeliness of the particular instance of UTRB that is used. As we have seen with UTRB2, in executions that actually deliver a message, some UTRB algorithms can exhibit "early stopping" in the sense that Δ_b is proportional to f, the number of processes that actually crash. However, when Δ_b is used to set a timeout for the delivery of a decision, it has to be instantiated using F rather than f to obtain the worst-case delay. Thus, when an ABORT decision occurs due to a timeout because no decision message was delivered, the delay is proportional to F. In other words, there may be executions were a single failure occurs (the coordinator crashes before sending any messages) yet no participant can decide before waiting for a time proportional to F. In this appendix we present a broadcast algorithm that can be used within a commitment protocol to expedite decisions that normally would have waited for the timeout period to expire in ACP-UTRB.

The algorithm we are about to present is motivated from the observation that participants gain knowledge about the imminent broadcast of a decision value during the voting phase [16]. In particular, a participant that received a VOTE_REQUEST message knows that a broadcast (of the decision) is supposed to be performed by the coordinator within 2δ time units. We can use this knowledge to obtain a "Terminating" version UTRB, called *Uniform Timed Terminating Reliable Broadcast* (UTTRB), that is able to detect that the broadcaster is faulty without having to wait for a time proportional to F. As usual, we first specify the properties of UTTRB and then develop an algorithm that achieves them.

Let $\mathcal{M}$ denote the set of messages that may be broadcast (in our case, $\mathcal{M}$ ={COMMIT, ABORT}). The set of messages that may be delivered is $\mathcal{M} \cup \{BF\}$, where $BF \notin \mathcal{M}$ is a special message indicating that the broadcaster is faulty.

UTTRB is specified by the same four properties B1–B4 of UTRB except with the following two modifications:

B2′ (Integrity): Each process delivers at most one message, and if it delivers $m \neq BF$ then some process must have broadcast m.

B3′ (Termination): If some correct process knows that a broadcast is supposed to start by some known local time, then every correct process eventually delivers some message $m \in \mathcal{M} \cup \{BF\}$ for that broadcast.

Note that since the broadcast is not allowed to broadcast BF, the Validity and Integrity properties of UTTRB imply that a process delivers BF only if the broadcaster is indeed faulty. Also note the the Termination property no longer has a delay parameter Δ_b since UTTRB guarantees delivery of a message as long as some process knows that a broadcast is supposed to happen.

Figure 10 illustrates an algorithm UTTRB1 that achieves the above properties.

Theorem 9 *UTTRB1 satisfies the Validity, Integrity, Termination and Uniform Agreement properties of UTTRB.*

Proof: To be filled in.

Use of UTTRB as the broadcast primitive in an Atomic Commitment Protocol requires very few changes to the general structure. Such a protocol is illustrated in Figure 11. Note that given the Termination property of UTTRB, the wait for the decision message does not need a timeout. Also, BF is now a possible delivery value and results in an ABORT decision in the case the coordinator crashes before sending a decision message to any participant.

Theorem 10 *UTTRB-ACP satisfies properties AC1–AC5 of (non-blocking) Atomic Commitment.*

Proof: The proofs of AC2 and AC4 are exactly as with ACP-UTRB.

(AC1) As before, the proof is by contradiction. Suppose p decides COMMIT while q decides ABORT. By AC4, $p \neq q$. Note that q can decide ABORT only in lines 14, 19 and 22. The proof that q cannot decide ABORT in lines 14 or 22 is exactly as in ACP-UTRB. We now show that q cannot decide ABORT in line 22, that is by delivering a BF or ABORT message.

By AX1, there is at most one participant that may assume the role of coordinator. By the protocol, this implies at most one broadcast of the decision is ever performed. Since p decides COMMIT (line 20), it must have delivered a COMMIT message. By the Uniform Agreement property of the broadcast, all correct participants, including q, eventually deliver this COMMIT message as well. From the Integrity property of the broadcast (and the fact that at most one broadcast regarding the decision is performed), q delivers at most one decision message. Thus q never delivers BF or ABORT.

(AC3) Suppose all participants vote YES and no failures occur. Let t_{start} be the real-time at which the transaction begins (this is the time at which some participant sends a T_START message). From AX2 and AX3, one participant assumes the role of coordinator by real-time $t_{start} + \Delta_c$. This coordinator immediately sends VOTE_REQUEST messages which arrive by $t_{start} + \Delta_c + \delta$. Note that in line 11 each participant waits for this VOTE_REQUEST, with a timeout set to trigger $\Delta_c + \delta$ time units after the real-time, t_{know}, it first learned about the transaction (in line 2 of Figure 1)[9]. In other words, this timeout is set to trigger at real-time $t_{know} + \Delta_c + \delta$. Since $t_{start} \leq t_{know}$, each participant receives the VOTE_REQUEST message it was waiting for, before the timeout is ever triggered. Thus, all participants send their YES votes to the coordinator. These votes arrive at the coordinator within 2δ time units of his sending the VOTE_REQUEST. Therefore, the timeout associated with the coordinator's wait for votes in line 3 never triggers. So, the coordinator receives YES votes from all participants and broadcasts a COMMIT message to all participants by real-time $t_{start} + \Delta_c + 2\delta$.

Note that in line 17, all correct participants are waiting for the delivery of this decision message. This wait has no timeout. From Figure 1 and the fact that the invoker does not crash, each correct participant receives T_START, and thus knows that a broadcast (about the decision) is supposed to start by local time $C_{know} + \Delta_c$. By the Termination and Validity

[9] Note that the variable C_{know} records the local time at which this event occurs.

```
procedure broadcast(m, $\mathcal{G}$)

% Broadcaster executes:
    send [MSG: m, cohorts, 1] to all processes in $\mathcal{G}$
    send [DLV: m] to all processes in $\mathcal{G}$

% Process p in $\mathcal{G}$ that knows a broadcast is supposed to start by local time $C_p$ executes:
    if (by local time $C_p + \delta$ no message [MSG: -, -, -] was received) then
        simulate receipt of [MSG: BF, cohorts, 1]

% Process p in $\mathcal{G}$ executes:
    upon (first receipt of (a possibly simulated) [MSG: estimate, cohorts, index])
        e, i := estimate, index
        first_timeout := local_clock + $(\delta + \tau)$
        for k := $0, 1, \ldots$ do
            set-timeout-to first_timeout + k$(2\delta + \tau)$
            wait-for (receipt of [DLV: estimate])
                deliver estimate
                exit loop
            on-timeout
                if (i $<$ F+ 1) then
                    i := i + 1
                    send [REQ: e, cohorts, i] to cohorts[i]
                    set-timeout-to local_clock + $2\delta$
                    wait-for (receipt of [MSG: estimate, cohorts, index])
                        e := estimate
                    on-timeout
                        skip
                else skip                                   % More than F cohorts have failed
        od

    upon (first receipt of [REQ: estimate, cohorts, index])
        send [MSG: estimate, cohorts, index] to all processes in $\mathcal{G}$
        send [DLV: estimate] to all processes in $\mathcal{G}$
end
```

Figure 10. UTTRB1: A Uniform Timed Terminating Reliable Broadcast Algorithm

```
procedure atomic_commitment(transaction, participants)
    cobegin
    % Task 1: Executed by the coordinator
        send [VOTE_REQUEST] to all participants                    % Including the coordinator
        set-timeout-to local_clock + $2\delta$
        wait-for (receipt of [VOTE: vote] messages from all participants)
            if (all votes are YES) then
                broadcast (COMMIT, participants)                           % Using a UTTRB
            else broadcast (ABORT, participants)                           % Using a UTTRB
        on-timeout
            broadcast (ABORT, participants)                                % Using a UTTRB

//

    % Task 2: Executed by all participants (including the coordinator)
        set-timeout-to $C_{know} + \Delta_c + \delta$
        wait-for (receipt of [VOTE_REQUEST] from coordinator)
            send [VOTE: vote] to coordinator
            if (vote = NO) then
                decide ABORT
            else
                % In contrast to ACP-UTRB, the wait for a decision has no timeout!
                wait-for (delivery of decision message)
                    if (decision message is BF or ABORT) then
                        decide ABORT
                    else decide COMMIT
        on-timeout
            decide ABORT
    coend
end
```

Figure 11. ACP-UTTRB: Non-Blocking Atomic Commitment Protocol Based on UTTRB

properties of the broadcast, every participant eventually delivers the COMMIT message. Thus all participants eventually decide COMMIT.

(AC5) If a correct participant executes UTTRB-ACP, it must have received T_START from the invoker and thus knows that a broadcast (about the decision) is supposed to start by local time $C_{know} + \Delta_c$. By the Termination property of the broadcast, it eventually delivers some message m ∈{COMMIT, ABORT, BF}. Thus, every correct participant that executes UTTRB-ACP eventually decides. □

B Coping with Imperfect Local Clocks

Up to now, we have assumed that local clocks of processes are perfect in that they run exactly at the same rate as real-time. In practice, local clocks only guarantee a bounded drift rate with respect to real-time. In other words, there exists a parameter $\rho > 0$ such that for all $t_2 \geq t_1$,

$$(1+\rho)^{-1}(t_2 - t_1) \leq C_i(t_2) - C_i(t_1) \leq (1+\rho)(t_2 - t_1)$$

where $C_i(t)$ is the reading of the local clock of process p_i at real-time t. Thus, local clocks are within a linear envelope of real-time. We assume that parameter ρ is common to all of the clocks.

In our protocols, clocks are used only to measure the passage of local time intervals in implementing the timeouts associated with various wait events at a process. In particular, they are never used in a manner where the clock value of process p_i is interpreted in the context of another process p_j. Thus, local clocks need not be synchronized with each other.

Since all of the parameters used by our protocols (e.g., δ, τ, Δ_c and Δ_b) are given in terms of real-time, we need to convert timeout periods specified in terms of real-time to local clock time. We also need to be able to convert a time interval measured by one process into an interval measured at another process. Note that these modifications are not exclusive to our protocols — *any* protocol that bases its actions on the passage of real-time must be modified in a similar manner in order to function correctly with realistic clocks. Our modifications are based on the following two observations:

- To guarantee that at least T seconds of real-time elapse, $T(1+\rho)$ ticks must elapse on a local clock,
- To guarantee that at least X ticks elapse on the local clock of process p_i, $X(1+\rho)^2$ ticks must elapse on the local clock of process p_j.

All of the algorithms and protocols we have developed remain correct if the timeout values are modified as follows based on the above observations:

Figure 2: Line 2 replace 2δ with $2\delta(1+\rho)$. Line 10 replace $\Delta_c + \delta$ with $(\Delta_c + \delta)(1+\rho)$. Line 16 replace $(\Delta_c + 2\delta + \Delta_b)$ with $(\Delta_c + 2\delta(1 + \ rho)^2 + \Delta_b)(1+\rho)$.

Figure 4: Same changes as in Figure 2.

Figure 6: Line 10 replace 2δ with $2\delta(1+\rho)$.

Figure 8: Line 5 Replace $(\delta + \tau)$ with $(\delta + \tau)(1+\rho)$. Line 7 Replace $k(2\delta + \tau)$ with $k(2\delta + \tau)(1+\rho)$.

Table 1: Case $f = 0$: No changes. Case $1 \leq f \leq F$: For $f = 1$ the correct expression is $\delta + (3\delta + 2\tau)(1+\rho)$. For $f > 1$ the correct expression is $\delta + (\delta+\tau)(1+\rho) + f(2\delta+\tau)(1+\rho) = \delta + f(3\delta + 2\tau)(1+\rho)$.

Figure 9: Lines 5, 7, 18 multiply all time constants by $1+\rho$.

Table 2: Case $f = 0$: No changes. Case $1 \leq f \leq F$: $\delta + (2\delta + 2\tau + f\delta)(1+\rho)$.

Acknowledgments We are grateful to Tushar Chandra for his comments on an early draft of this work.

References

[1] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Systems*, Montreal, Canada, August 1983, 27–30.

[2] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, vol. 13, no. 2, June 1981, 185–222.

[3] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[4] T. Chandra and S. Toueg. Time and Message Efficient Reliable Broadcast. In *Proc. of the 4th International Workshop on Distributed Algorithms*, September 1990, Bari, Italy, J. van Leeuwen and N. Santoro (Eds.), Lecture Notes in Computer Science, vol. 486, Springer-Verlag, 289–300. Full version available as Cornell Technical Report, TR 90-1094, May 1990.

[5] B.A. Coan and J. Lundelius. Transaction Commit in a Realistic Fault Model. In *Proc. of the 5th ACM Symp. on Principles of Distributed Systems*, Calgary, Alberta, Canada, August 1986, 40–51.

[6] D. Dolev and H.R. Strong. Distributed Commit with Bounded Waiting. In *Proc. of the 2nd Symp. on Reliability in Distributed Software and Database Systems*, 1982, 53–60.

[7] C. Dwork and D. Skeen. The Inherent Cost of Non-Blocking Commitment. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Systems*, Montreal, Canada, August 1983, 1–11.

[8] M. Fischer, N. Lynch and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, vol. 32, no. 2, April 1985, 374–382.

[9] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. on Computers*, vol. C-31, no. 1, January 1982, 48–59.

[10] H. Garcia-Molina, F. Pittelli and S. Davidson, Applications of Byzantine Agreement in Database Systems. Technical Report TR 316, Princeton University, Princeton, New Jersey, June 1984.

[11] J.N. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham and G. Seegmuller (Eds.), Lecture Notes in Computer Science, vol. 60, Springer-Verlag, 1978.

[12] J.N. Gray. A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem. In *Fault-Tolerant Distributed Computing*, B. Simons and A. Z. Spector (Eds.), Lecture Notes in Computer Science, vol. 448, Springer-Verlag, New York, 1990, 10–17.

[13] V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. Ph.D. dissertation, Harvard University, 1984.

[14] V. Hadzilacos. On the Relationship Between the Atomic Commitment and Consensus Problems. In *Fault-Tolerant Distributed Computing*, B. Simons and A. Z. Spector (Eds.), Lecture Notes in Computer Science, vol. 448, Springer-Verlag, New York, 1990, 201–208.

[15] V. Hadzilacos and S. Toueg. Reliable Broadcast and Agreement Algorithms. In *Distributed Systems* (Second Edition), S. J. Mullender (Ed.), ACM Press, New York, 1993.

[16] J.Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM*, vol. 37, no. 3, July 1990, 549–587.

[17] L. Lamport. The Part-Time Parliament. Technical Report 49, DEC Systems Research Center, Palo Alto, California, September 1989.

[18] L. Lamport and P.M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, vol. 32, no. 1, January 1985, 52–78.

[19] L. Lamport and M.J. Fischer. Byzantine Generals and Transaction Commit Protocols. Computer Science Laboratory, SRI International, Op. 62, 1982.

[20] L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. *ACM Trans. Programming Languages and Systems*, vol. 4, no. 3, July 1982, 382–401.

[21] B. Lampson. Atomic Transactions. In *Distributed Systems Architecture and Implementation: An Advanced Course*, B. Lampson, M. Paul and H. Siegert (Eds.), Lecture Notes in Computer Science, vol. 105, Springer-Verlag, 1981, 246–265.

[22] C. Mohan, R. Strong and S. Finkelstein. Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement within Clusters of Processors. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Systems*, Montreal, Canada, August, 1983, 89–103.

[23] G. Neiger and S. Toueg. Automatically Increasing the Fault-Tolerance of Distributed Algorithms. *Journal of Algorithms*, vol. 11, no. 3, September 1990, 374–419.

[24] C.H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, vol. 26, no. 4, October 1979, 631–653.

[25] D. Skeen. *Crash Recovery in a Distributed Database System*. Ph.D. dissertation, University of California at Berkeley, 1982.

[26] D. Skeen. Determining the Last Process to Fail. *ACM Trans. on Computer Systems*, vol. 3, no. 1, February 1985, 15–30.