THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE VERSAILLES

Spécialité

INFORMATIQUE

Présentée par

Maha ABDALLAH

Pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE VERSAILLES

Sujet de la thèse

DICTATORIAL TRANSACTION PROCESSING: FROM HIGH-PERFORMANCE TO FAULT-TOLERANCE

Soutenue le 27 mars 2001 devant le jury composé de

Pr. Amr EL ABBADI	University of California at Santa Barbara	Examinateur
Pr. Jean FERRIÉ	Université de Montpellier II	Rapporteur
Pr. Georges GARDARIN	e-XMLmedia	Examinateur
Pr. Rachid GUERRAOUI	EPFL	Président
Pr. Philippe PUCHERAL	Université de Versailles St-Quentin	Directeur
Pr. Patrick VALDURIEZ	Université Pierre & Marie Curie - INRIA	Rapporteur

Université de Versailles St-Quentin en Yvelines – Laboratoire PR*i*SM

Acknowledgements

I am deeply grateful to several people who contributed in various ways to this work. First and foremost, I wish to express my gratitude to my advisor, Prof. Philippe Pucheral, for guiding my first steps through the research world, for sharing with me his research enthusiasm, for his patience and friendliness, and for expressing his confidence in me through a great freedom of action.

I am also very grateful to Prof. Georges Gardarin for accepting me in his research group, and for sharing with me his teaching experience by giving me the opportunity to work as a teaching assistant in his database courses.

Special thanks go to Prof. Rachid Guerraoui for the great interest he has shown in my work. Through his continuous help, and the invaluable scientific discussions we had, Rachid was involved in almost every part of this work. Without his contributions, Chapter 3 would probably not have existed.

I wish to express my gratitude to Prof. Jean Ferrié, and Prof. Patrick Valduriez, for the time they have spent applying their expert knowledge to examining and reporting on this thesis. I am also deeply grateful to Prof. Amr El Abbadi for accepting to serve on my thesis committee, and for giving me the opportunity to pursue a postdoctoral work in his research group at the Department of Computer Science, University of California, Santa Barbara.

My sincere thanks go to the TransRep project members at INRIA, in particular Dr Simone Sédillot and Dr Malik Saheb, for their contributions to the integration of the 1PC idea into OTS, and for helping me understand the different facets of MAAO-OTS.

My warm thanks go to Khaled Boussetta, Catherine Blirando, and Mourad Guerroui for their friendship, and for lending me a sympathetic ear each time I needed someone to talk to. I am also extremely thankful to Thierry and Yvette, my French family, for their faithful and loving support that helped me carry through this research.

Finally, I am grateful to my brother, Nizar, and my sister, Hitaf, for being my greatest source of strength, and such a positive influence on my life. There is much of your love and caring support behind these pages.

Last, but definitely not least, I wish to express my deepest gratitude to my parents for just being the great persons they are. This thesis is simply the result of their unconditional love and devotion. The rest cannot be put into words !

To the one who taught me how to think,
to Bassima…

Contents

Chapter 1

Introduction

1.1 Research Context

Over the past two decades, *distributed systems* have become commonplace in several computing domains. With the recent advances in communication systems, the explosion of the Internet, and the now ubiquitous *World-Wide Web* (WWW), not only is the computing infrastructure changing, but also the user community is underlying a similar revolution. Distributed systems seem to be everywhere in our daily life activities, making them a concern of almost every individual.

As we depend more and more heavily on distributed systems and applications, the *reliability* of these becomes increasingly critical. Reliability is particularly difficult to tackle in a distributed environment since we have to deal with some of the intrinsic characteristics of distribution, notably partial failures or unreliable communication. Reliability generally connotes two fundamental properties: *safety* and *liveness* [Lam77, AlS85, Gue96][1]. Roughly speaking, a safety property stipulates that "*bad things do not occur*" during execution. In information systems, for instance, the proscribed "bad thing" would be the violation of *data consistency*. In this context, the safety of an application expresses its ability to maintain the consistency of accessed data objects even in the event of failures or concurrent executions. A liveness property stipulates that "*eventually good things do occur*" during execution. The desirable "good thing" can express requirements like *state progress*, *program termination*, or *service availability*.

[1] Safety and liveness properties were first introduced by Lamport in [Lam77], and have been since adopted as the usual metrics to evaluate the reliability degree of distributed systems.

1.1.1 Transaction Processing

In all information systems, as for database management systems, telecommunication systems, industrial control systems, finance, or even electronic commerce, preserving data consistency (i.e., applications' safety) in the presence of failures or concurrent data accesses relies on the *transaction* concept. Transactions are powerful abstractions that enable the structuring of distributed systems in a reliable manner, while relieving the programmer from dealing with the complexity of concurrent programming or failures.

A transaction is an atomic set of operations updating shared data objects and satisfying the so-called ACID properties [GrR93, BCF97], namely *atomicity*, *consistency*, *isolation*, and *durability*. In a distributed transactional system, a transaction may access shared data objects residing at multiple sites. A *distributed transaction* is decomposed into one transaction *branch* per accessed site. Even though it is generally assumed that each site where a distributed transaction executed ensures the local ACID properties of its transaction branch, the atomicity and isolation of a distributed transaction can be jeopardized in the absence of a global control. Therefore, some additional measures must be taken so that global atomicity and global isolation of distributed transactions are guaranteed.

1.1.2 Atomic Commitment

This thesis deals with the global atomicity problem, which requires that either all the updates performed by the transaction on the different accessed sites are made permanent, or all of them are obliterated. Since each local site participating in the transaction execution ensures the local ACID properties of its transaction branch, the task of ensuring the global atomicity of a distributed transaction reduces to ensuring that the transaction either commits at all the sites, or it aborts at all the sites. To solve this distributed agreement problem, known as the *Atomic Commitment* (AC) problem [BHG87], every participant expresses through a *vote* its ability to make its updates permanent, and all participants need to agree on a unique outcome (*commit* or *abort*) for the transaction. A protocol that achieves this kind of agreement is called an *Atomic Commitment Protocol* (ACP).

1.2 Research Motivations

This work originated from the firm conviction that although the atomic commitment problem has been intensively studied in the last two decades, it remains in perpetual mutation to adapt to today's new environments and applications. As this thesis testifies, existing solutions to the problem suffer from their lack of flexibility with respect to the distributed computing technology revolution in the sense that they can no longer meet the requirements of today's distributed systems and applications. Indeed, the simplest and best-known ACP on which rely existing systems to coordinate transaction commitment is the *Two-Phase Commit* (2PC) protocol [Gra78, BHG87]. Although widely used and de facto standard [OMG00a, X/Op91, ISO92a], 2PC suffers from three major drawbacks when employed in the context of today's distributed systems and applications:

- It is quite *inefficient* in terms of both time delay and message complexity. This is mainly due to the number of communication steps and forced log writes needed in order to commit a transaction even in the absence of failures. This inefficiency not only makes 2PC inadequate to today's highly reliable distributed platforms, but also is particularly unacceptable in advanced and critical applications, such as Supervision and Control Systems' applications (SCS)[2] [ABG98], with strong *performance* requirements.

- It may lead to *blocking* situations in which operational sites are prevented from terminating the transaction due to failures in other components of the system [Ske81]. During these blocking periods, operational sites are also prevented from releasing valuable system resources they may have acquired for exclusive use on behalf of the transaction (otherwise transaction *safety* would be compromised), thereby compromising transaction *liveness*, and hence system *availability*. Although this situation might be acceptable for some standard applications, in mission critical applications however (e.g., SCS applications), for which a *short response time* is a crucial factor, some liveness guarantees are indispensable. Similar requirements

[2] Typical examples of SCS are command and control systems in the field of transport (air, railway and road) including traffic management and fleet management systems, technical management systems for large equipment infrastructures such as telecommunication networks or electricity and water distribution networks.

arise in applications involving an important number of sites (e.g., Internet applications) where it would be completely unconceivable to block the entire system due to the crash of one single site. Protocols that provide liveness guarantees despite concurrency and failures are called *non-blocking* protocols (also known as *fault-tolerant* protocols).

- It forces participants in a transaction to externalize a local *prepared state*. The consequence of this is threefold. First, it violates *site autonomy*[3], precluding the integration of legacy systems [ShL90] in distributed transactions. Second, it consumes valuable system resources on data servers hosted by lightweight intelligent devices with very limited resources, such as palmtops, cellular phones, or even smart cards [BPA00, BBP00]. Third, it leads to the abort of a transaction after it has been successfully processed if any of its participants is unreachable during the first phase of the protocol. The impact of this behavior is exacerbated in mobile environments in which (accidental or voluntary) disconnections are very frequent [BPA00].

Several optimized variations and non-blocking alternatives to 2PC have been proposed in the literature [Ske81, Ske82, MLO86, StC90, StC93, LaL93, BaT93, KeD94, AlC95, GuS95, GLS95, AlC96, GLS96]. However, none of these protocols is able to combine efficiency during normal processing with fault-tolerance (i.e., non-blocking), or to consider the issue of local site autonomy. Given these limitations, the need for a novel solution to the distributed commit problem that is capable of reconciling such crucial yet antagonistic requirements becomes an unquestionable fact. In this thesis, we have sought to address this issue.

1.3 Research Contributions

The major objective of this work is to bridge the gap between performance and fault-tolerance of atomic commitment protocols, while considering the challenging and key aspect of today's large distributed environments, namely local site autonomy. Another

[3] Site autonomy means that (1) participants' local information (e.g., log records or lock tables) cannot be externalized, and (2) no changes can be made to the participating sites to accommodate the distributed system.

important objective is the compliance of the proposed solutions with current transactional standards, initially designed with 2PC in mind.

1.3.1 Atomic Commitment: Performance

As its name indicates, 2PC (and its variations) is made out of two phases. In the first phase, called the *voting* phase, the participants are given an ultimate right to abort the transaction (i.e., the *veto right*), and in the second phase, called the *decision* phase, the participants need to agree on the same decision (*commit* or *abort*). Whereas the *decision* phase is indeed necessary to ensure transaction *atomicity* (otherwise the participants might disagree on the transaction's outcome), one might wonder whether the *voting* phase can (sometimes) be eliminated. This would drastically reduce the cost of commitment (two communication steps together with their associated forced log writes would be gained), and participants would not need to externalize a local prepared state anymore. Roughly speaking, to commit a transaction, the coordinator of the commit protocol would simply need to force-write the decision and send one message to the participants.

The idea of *One-Phase Commit* (1PC) is not new: it was informally discussed by Gray in [Gra78, Gra90] as well as by Stonebraker in [Sto79]. More recently, several 1PC variations have been suggested in the literature [StC90, StC93, AlC95, AlC96]. Despite their efficiency, 1PC protocols have been completely ignored in the implementation of distributed transactional systems. We believe that the reason for this is due to some (strong) assumptions made by 1PC protocol designers about the underlying transactional systems without any statement on the necessity of those assumptions. This gives the impression, from a practical point of view, that 1PC is just an exotic concept with unrealistic underlying assumptions and, from the theoretical point of view, that 1PC does not make any sense as it contradicts proven lower bounds on the cost of solving the atomic commitment problem in distributed systems [DwS83].

This work started with the broad objective of identifying the assumptions under which 1PC can be used. To our knowledge, none of the previous works that were devoted to 1PC either defines the abstract properties of the problem that is solved or gives a precise description of the impact of eliminating the *voting* phase on transaction processing. In this

context, the present thesis provides three major contributions: it introduces the Dictatorial Atomic Commitment problem, defines On-line Serializability and On-line Commit-Resiliency, and proposes the Coordinator Logical Log mechanism.

Dictatorial Atomic Commitment. We point out the fact that removing the *veto right* from atomic commitment comes down to an agreement problem that is different from the traditional atomic commitment problem solved by a 2PC [BHG87]. In light of this observation, we give a precise abstract specification of the resulting problem, which we baptize the *Dictatorial Atomic Commitment* (DAC) problem, and propose a simple algorithm that solves it. A crucial feature of this algorithm is that it can be seen as the basic building block around which all existing 1PC variations are designed. The lack of the *veto right* explains why 1PC is actually more efficient than any of the well-known optimized variations of 2PC [MLO86].

On-line Serializability & On-line Commit-Resiliency. Given the abstract specification of the DAC problem, we investigate its impact on the concurrency control and recovery protocols employed by the participants in a transaction. In particular, we define three conditions that are necessary and sufficient to ensure the correctness of transactional systems with no participant *veto right*: *on-line serializability*, *cascadelessness* and *on-line commit-resiliency*. These conditions are strictly stronger than the usual correctness metrics for transactional systems, namely *serializability*, *recoverability* and *resiliency,* respectively [Had88]. We also discuss the practical impact of those conditions on real transactional systems, and show that unlike *on-line serializability* and *cascadelessness*, *on-line commit-resiliency* is however rarely realistic in practice.

Coordinator Logical Log. Given the above limitation, we investigate techniques employed by existing 1PC protocols to circumvent the need for *on-line commit-resiliency* by considering "non-classical" atomic commitment schemes in which participants in a transaction are allowed to delegate part of their transactional responsibilities to the coordinator of the protocol. We point out the fact that although the existing techniques overcome *on-line commit-resiliency*, they come however at a very high cost as they

violate site autonomy, which compromises their use in existing commercial systems. We then study an adaptation of those techniques and propose a new 1PC variation, named *Coordinator Logical Log* (CLL), which preserves site autonomy, making 1PC indeed realistic and useful in practice.

1.3.2 Atomic Commitment: Fault-Tolerance

The second major part of this research deals with the non-blocking dictatorial atomic commitment problem. This problem is of major importance given that, compared to 2PC, 1PC increases the probability to blocking of participants in case of failures. Indeed, by removing veto rights from atomic commitment, the window of vulnerability to blocking of the protocol lasts all along the transaction. In this context, our work provides two major contributions: it proposes the Non-Blocking Coordinator Logical Log protocol, and the Asynchronous Non-Blocking Coordinator Logical Log protocol.

Non-Blocking Coordinator Logical Log. We propose a solution to the non-blocking dictatorial atomic commitment problem in the context of synchronous systems. The resulting protocol can be seen as a straightforward extension of CLL, called *Non-Blocking CLL* (NB-CLL), that achieves non-blocking based on a *Uniform Timed Reliable Broadcast* (UTRB) primitive and assuming reliable failure detection.

Asynchronous Non-Blocking Coordinator Logical Log. Obviously, the assumption of a synchronous system and a reliable failure detector is not always realistic in practice since variable or unexpected workloads are sources of asynchrony. Therefore, we propose a new non-blocking extension to CLL, called Asynchronous NB-CLL (ANB-CLL), that achieves non-blocking in an asynchronous system augmented with an unreliable failure detector, and in which processes may *crash* and *recover*. To our knowledge, it is the first time that the non-blocking atomic commitment problem is studied in the context of asynchronous systems based on a *crash-recovery* model of computation. An interesting feature of our non-blocking solutions is that they can be directly applied to any existing 1PC protocol. Performance analysis shows that NB-CLL and ANB-CLL are more efficient in terms of time delay, message complexity and

number of forced log writes than all other non-blocking commit protocols proposed in the literature. Furthermore, they appear to be the sole protocols that can cope with existing transactional systems without violating their autonomy.

1.3.3 Pragmatic Implementation

We are currently finalizing the implementation of the ANB-CLL protocol in the context of the OpenDREAMS-II project (Esprit-VI R&D project n° 25262) in which I have been participating since 1997. The project is financed by the European Union and aims at designing and building a CORBA compliant platform dedicated to industrial Supervision and Control Systems (SCS). The OpenDREAMS-II platform is augmented with several components and services specifically tailored to answer SCS requirements, including a Transaction Service designed and implemented by the PR*i*SM laboratory of the University of Versailles.

The project platform is experimented and validated through two industrial applications, namely a Condition Monitoring and Diagnostics of Thermal Power Plants application, as well as an Advanced Surface Movement Guidance & Control Systems (A-SMGCS) application for managing all moving vehicles in an airport environment. Both applications showed the effectiveness of our protocol in meeting SCS requirements in terms of performance and fault-tolerance. The implementation of the ANB-CLL prototype is at a far advanced stage that enables us to prove the validity of our theoretical study, and to show the compatibility of our protocol with existing transactional standards (OTS/CORBA, XA/DTP) and commercial database systems.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapters 2 and 3, which constitute the first major part of this work, tackle performance issues related to distributed commit protocols. In Chapter 2, we define a general model of a distributed transactional system that we follow throughout the thesis. We then give some background about the Atomic Commitment problem, and recall the Two-Phase Commit approach to the problem through a description of the most well-known 2PC variations commonly found in the

literature. We finally point out 2PC limitations in terms of performance and applicability to existing transactional systems.

In Chapter 3, we present proposals to overcome those limitations. We first introduce the Dictatorial Atomic Commitment (DAC) problem, resulting from removing veto rights from the traditional Atomic Commitment problem, and propose a highly efficient algorithm that solves it based on a One-Phase Commit (1PC) approach. We next define three necessary and sufficient conditions to ensure the correctness of transactional systems with no participant veto right: *on-line serializability*, *cascadlessness*, and *on-line commit-resiliency*, and discuss the practical impact of those conditions on concurrency control and recovery protocols. Based on this discussion, we draw an interesting parallel between existing 1PC variations, and point out their practical limitations. We finally propose the Coordinator Logical Log (CLL) protocol, a new 1PC variation that capitalizes on the existing ones so as to keep the best of the 1PC approach while being useful and practical.

Chapters 4 and 5, which constitute the second major part of this thesis, extend the work presented in the previous chapters on distributed commit protocols to tackle fault-tolerant issues. In Chapter 4, we recall the issue of blocking in 2PC, and define the Non-Blocking Atomic Commitment problem. We then present a survey of existing non-blocking commit protocols commonly found in the literature. In order to do so, we refine the general system model described in Chapter 2 in order to reflect different assumptions about failures and failure detections, and focus on the two extremes of a spectrum of possible models, namely *synchronous* and *asynchronous* systems. Each protocol is then described in the context of the underlying system model it assumes. We finally point out the limitations of the discussed protocols in terms of both performance and compliance with existing transactional systems.

In Chapter 5, we provide solutions to those limitations by extending our results on dictatorial transaction processing to cover fault-tolerance issues. We first discuss the blocking problem in 1PC and refine the Dictatorial Atomic Commitment problem specification to include the non-blocking property. We then propose the NB-CLL and ANB-CLL protocols that solve the problem in the context of synchronous and asynchronous systems, respectively. These protocols blend the efficiency of the One-Phase Commit approach with non-blocking, without compromising their practical applicability to existing commercial systems.

Chapter 6 constitutes the third and final part of this thesis. It briefly surveys existing distributed transaction processing standards, and discusses a practical prototype implementation of ANB-CLL in the context of the OMG's Object Transaction Service (OTS). Finally, Chapter 7 summarizes the major contributions of this thesis, and discusses some future research directions and open issues around this work.

Part I

Performance Issues

Chapter 2

Atomic Commitment: Background

A significant body of literature is available on distributed commit protocols. In order to put our work into perspective, we give in this chapter an overview of some of these protocols. The chapter is not intended to provide a complete survey on the matter but rather to highlight the essential by concentrating on well-established protocols that have received the most attention in the transactional world. In order to do so, we first define a general model of a distributed transactional system. We then recall some background about the Atomic Commitment (AC) problem, and discuss the basic Two-Phase Commit (2PC) protocol together with its best-known optimizations. We finally point out the limitations of the Two-Phase Commit approach in answering the needs of today's distributed systems and applications.

2.1 Distributed Transactional System Model

Distributed computing problems have been studied in a variety of computational models. In this section, we define a general model of a distributed transactional system that we follow throughout this thesis. In Chapters 4 and 5, we refine our model and make it more precise in order to reflect the different assumptions we make on the environment, and also on the failures and the failure detection mechanisms we consider.

2.1.1 Sites and Processes

We consider a distributed system composed of a finite set of sites $\Pi = \{S_1, S_2, ..., S_n\}$ completely connected through a set of communication channels[4]. Each site has a local memory and executes one or more processes. For the sake of simplicity, we assume only one process per site. We consider the so-called *message-passing* communication model in the sense that processes (sites) communicate with each other by exchanging messages. To simplify the subsequent discussion, when a process disseminates a message to every other process, we will speak as if the process sends the message to itself (and reacts accordingly).

At any given time, a process may be *operational* or *down*. While operational, a process is assumed to follow exactly the actions specified by the algorithm it is running. Operational processes may go down due to *crash failures* [LaF82], i.e., we do not consider *Byzantine failures* in which processes can behave arbitrarily [LSP82, Fis83]. A process is said to be *correct* if it has never crashed; otherwise, the process is said to be *faulty* [5].

We consider a *crash-recovery* failure model in the sense that a process can be down (crash) and later become operational again. When it does so, we say that the process *recovers,* in which case it executes a specific *recovery protocol*. A process that is down stops all its activities, including sending messages to other processes, until it recovers. Each process has access to a *stable storage* (i.e., that sustains crash failures) in which it maintains information necessary for the recovery protocol. During recovery, a process restores its local state based on the information it wrote on stable storage.

2.1.2 Transactions

A *transaction* is an atomic set of operations updating shared data objects and satisfying the so-called ACID properties [GrR93, BGS92, BCF97], namely *atomicity*, *consistency*, *isolation*, and *durability*. The *atomicity* property, also called *all-or-nothing* property, means that either the transaction successfully executes to completion and the effects of all of its operations are recorded in the accessed objects (the transaction is said to be

[4] Note that although the physical network is not always completely connected, virtual links between every pair of processes can be provided by network layer protocols.

[5] Note that the period of interest of these definitions is the duration of the commit protocol, i.e., a process is correct if it never crashes during the execution of the commit protocol.

committed), or it fails and it has no effect at all (the transaction is *aborted*). In other word, all the transaction's operations are treated as a single, indivisible, atomic unit. *Consistency* means that the transaction does not violate the integrity constraints of accessed shared objects, while *isolation* means that the intermediate effects of a transaction are not visible to concurrent transactions. *Durability* means that the updates of a committed transaction are permanent (i.e., stored on a stable storage that sustains failures).

The *atomicity* and *durability* properties have been formalized through the *resiliency* theory [BHG87, Had88], and are usually ensured using a set of protocols known as *recovery protocols*. *Isolation* has been formalized through the *serializability* and *recoverability* theories [BHG87, Had88], and is ensured using a set of protocols referred to as *concurrency control protocols*. *Consistency* is generally assumed to be the responsibility of the transaction programmer (i.e., users are required to write transactions such that each takes the database from one consistent state to another) and can be enforced by some semantic integrity mechanisms built into the system.

A *distributed transaction* (henceforth called a "transaction") accesses shared objects residing at multiple sites. For each transaction, the processes that perform operations on its behalf are called transaction *participants*. The portion of a transaction executed at one participant is called a transaction *branch*. In the following, we assume the "classical" distributed transactional scheme in the sense that each participant ensures the ACID properties of every transaction branch it executes. We also assume that for every transaction, there is one specific participant, called the transaction *coordinator*, which manages the transaction processing and termination[6].

The coordinator forwards every transaction operation to the participant hosting the object involved by the operation. If a participant succeeds in processing an operation, it replies by sending back an *acknowledgment* message; otherwise, the participant aborts the transaction and sends back a *negative acknowledgment*. To conclude the transaction, the coordinator triggers an *Atomic Commitment Protocol* (ACP) whose aim is to ensure that a logical atomic *commit* or *abort* action is consistently carried out at all participants despite failures. In the following, we recall the abstract formulation of the underlying agreement problem.

[6] This is generally the site where the transaction originated.

2.2 The Atomic Commitment Problem

The *Atomic Commitment* (AC) problem is a distributed agreement problem that is concerned with bringing all participants in a transaction to agree on a unique outcome (*commit* or *abort*) for that transaction. This problem was formally defined in [BHG87]. Each participant has exactly one of two votes: *yes* or *no*, and can reach exactly one of two decisions: *commit* or *abort*, such that the following properties are satisfied:

- **AC-Uniform-Agreement:** No two participants reach different decisions.
- **AC-Uniform-Validity:** *commit* is decided only if all participants vote *yes*.
- **AC-Uniform-Integrity:** No participant can reverse its decision after it has reached one.
- **AC-Non-Triviality:** If all participants vote *yes* and no failures occur, then all participants must decide *commit*.

The vote of a participant reflects its ability to commit its transaction branch. A participant votes *yes* only if the local execution of its transaction branch was successful and it is *ready* and *willing* to make its updates permanent even in the presence of failures. This actually means that the participant can locally guarantee the ACID properties of its transaction branch. A *no* vote (or *abort*) indicates that due to some local problems (integrity constraint violation, concurrency control problem, memory fault or storage problem), the participant is not able to guarantee some of the ACID properties of its transaction branch. An ACP is an algorithm that satisfies all of the four properties of the AC problem.

The *AC-Uniform-Agreement*, *AC-Uniform-Validity* and *AC-Uniform-Integrity* conditions are safety conditions in the sense that they ensure the atomicity property of the transaction. The *AC-Non-Triviality* condition excludes from consideration trivial solutions to the problem in which participants always decide *abort*.

2.3 The Basic Two-Phase Commit Protocol

All 2PC variations can be regarded as optimizations to the basic 2PC protocol. In this section, we recall the principle of the Two-Phase Commit approach in general, and discuss the details of the basic 2PC protocol in particular.

2.3.1 Failure-Free Execution

The basic *Two-Phase Commit* (2PC) protocol [Gra78, BHG87] (together with its variations) solves the AC problem by performing a *voting phase* and a *decision phase*. In the voting phase, the coordinator sends a *request-for-vote* message (also called a *prepare* message) to all the participants in the transaction. Each participant replies by sending its vote. If a participant votes *yes*, it enters a *prepared state* during which it can neither commit nor abort the transaction unless it receives the final decision from the coordinator. The period of time from the moment a participant votes *yes* and until it receives the final decision is called the *uncertainty period* for that participant. If, on the other hand, a participant votes *no*, it can unilaterally abort its transaction branch.

During the decision phase, the coordinator decides on the transaction depending on the votes it receives from the participants. If all participants have voted *yes*, the coordinator decides *commit,* and sends its decision to all the participants in the transaction. Otherwise, the coordinator decides *abort,* and sends its decision (only) to the participants that are in the prepared state, i.e., those participants that voted *yes*. When a participant receives the final decision, it complies with this decision and sends back an *acknowledgment* message. This acknowledgment is a promise from the participant that it will never ask the coordinator about the outcome of the transaction. Finally, after receiving acknowledgments from all the prepared participants, the coordinator can forget about the transaction. This describes 2PC assuming no failures occur during the protocol execution. It is easy to see that 2PC satisfies all of the four properties of the AC problem.

2.3.2 Dealing with Failures

In order to exclude uninteresting protocols that allow participants to remain undecided forever once some failures have occurred during the protocol execution, the following *AC-Termination* property has to be added to the specification of the AC problem [BHG87].

- **AC-Termination:** If all failures are repaired, then unless a new failure occurs, every participant eventually reaches a decision.

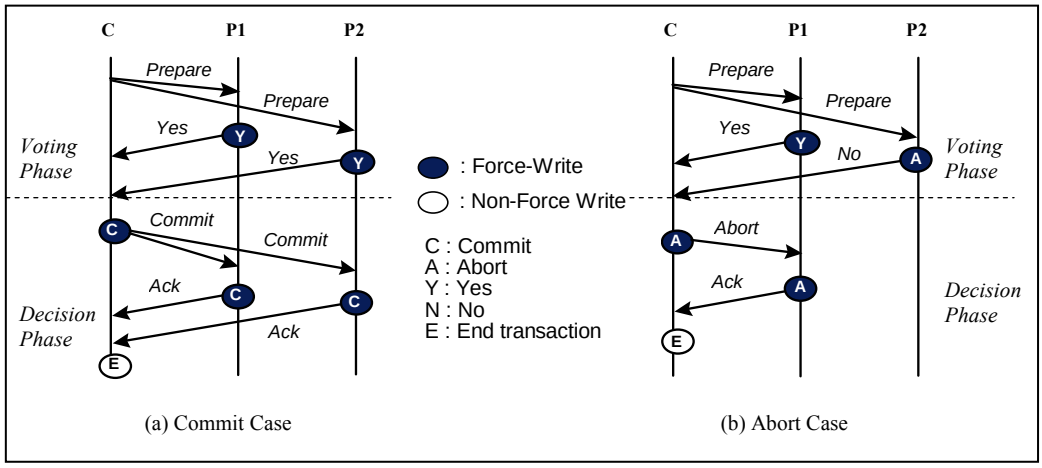

Figure 2.1: *The basic 2PC protocol*

To satisfy *AC-Termination*, specific actions that deal with site and communication failures must be supplied. First, failures may prevent one site from communicating with another, leading a process to wait indefinitely for a message that may never arrive. To avoid such a situation, special timeout actions must be associated with each point in the protocol where a process is waiting for a message. Furthermore, since we consider a *crash-recovery* failure model, participants can be down and later become operational again. In this case, a recovering process must attempt to reach a decision consistent with the decision operational processes may have reached. In the following, we consider these two issues in turn.

Timeout Actions

There are three cases to consider: (1) a participant is waiting for the *prepare* message from the coordinator, (2) the coordinator is waiting for participants' *vote*s, and (3) a participant is waiting for the coordinator's decision.

Case (1): If a participant P_i times out waiting for the prepare message from the coordinator, P_i can unilaterally decide abort since it has not voted yet.

Case (2): If the coordinator times out waiting for a participant vote, it can safely decide abort. This is because at this point, the coordinator has not reached any decision yet, and no participant can have decided commit.

Case (3): If a participant P_i times out waiting for the decision message (i.e., while in its uncertainty period), it cannot decide on its own. In this case, P_i starts a *termination protocol* during which it tries to find out what to decide by contacting another participant that either (i) knows the decision, or (ii) can unilaterally decide on the transaction. If, however, all participants with which P_i can communicate neither satisfy (i) nor (ii), P_i remains blocked until it can communicate with at least one such participant. When used with 2PC, this termination protocol satisfies the *AC-Termination* property. Indeed, if all failures are repaired, and no new failures occur, P_i will eventually be able to communicate with a participant for which either (i) or (ii) holds, namely the coordinator.

Crash Recovery

Recovery is made possible by recording the progress of the protocol during normal processing (i.e., in the absence of failures) in the logs of the coordinator and the participants. Since failures can occur at any time, some of the information stored in the logs must be *force-written*, i.e., written immediately to a stable (nonvolatile) storage that sustains failures. For instance, the coordinator force-writes its decision before sending it to the different participants. Each participant force-writes (1) its vote before sending it to the coordinator, and (2) the final decision before acknowledging the coordinator. Usually, a participant that votes *yes* force-writes its vote together with all the updates performed on behalf of the transaction. This ensures that the participant's updates are permanent even if it crashes (i.e., to ensure transaction *resiliency*). Force-writing a decision record in the log is the act by which a process *decides* on the transaction. When the coordinator receives acknowledgments from all participants, it writes a non-forced *end* record, indicating that the information pertaining to the transaction can be garbage collected from its log. Finally, it is important to note that forcing a log record implies that the forced log record *and all* preceding (non-forced) ones are moved immediately from main memory buffers to stable storage. Figure 2.1 describes the protocol execution between a coordinator C, and two participants P_1 and P_2.

Consider a participant P_i recovering from a crash. A failed participant returns to the operational state by executing a *recovery protocol*. During this protocol, P_i first restores a consistent local state using the information it stored in its stable log. Then, it tries to

decide on the transactions that were active at the time the crash occurred (i.e., transactions for which no decision record exists in the log). For each of these transactions, if P_i does not find a *yes* record in its log, it can unilaterally decide *abort*. If, on the other hand, a *yes* record is found, this means that P_i failed while in its uncertainty period, and therefore, P_i is exactly in the same state as if it had timed out waiting for the decision message. Thus, the termination protocol described above can be used to decide on the transaction.

2.4 2PC Optimizations

The efficiency of an atomic commitment protocol is usually measured following three performance metrics [BHG87, MLO86, AbP97]: (1) *message complexity*, which corresponds to the number of coordination messages that need to be exchanged between the participants in the transaction, (2) *time complexity*, which corresponds to the number of communication steps or rounds required until a decision is reached at every participant, and (3) *log complexity*, which corresponds to the number of forced log writes performed by the participants in order to support recovery. The latter is of particular importance since it determines the number of blocking I/O required for a good behavior of the protocol.

As already mentioned in Chapter 1, 2PC introduces a considerable latency in the system even in the absence of failures. Assuming that n is the total number of participants in the transaction, 2PC requires 3 communication steps (*request-for-vote*, *vote* and *decision*) and $2n+1$ forced log writes until a decision is reached at every participant[7]. The higher the latency of an ACP, the longer the length of time a transaction may be holding shared objects, preventing other transactions from accessing these objects. Furthermore, 2PC has a high message complexity due to $4n$ messages (including the acknowledgement of the decision) exchanged during the protocol execution. These significant overheads have motivated many researchers to propose several optimizations to the basic 2PC.

[7] Note that in all our evaluations, and in accordance with Section 2.1.1, we assume that when the coordinator sends a message to all participants in the transaction, it also sends the message to itself, and acts, just like any other participant, accordingly.

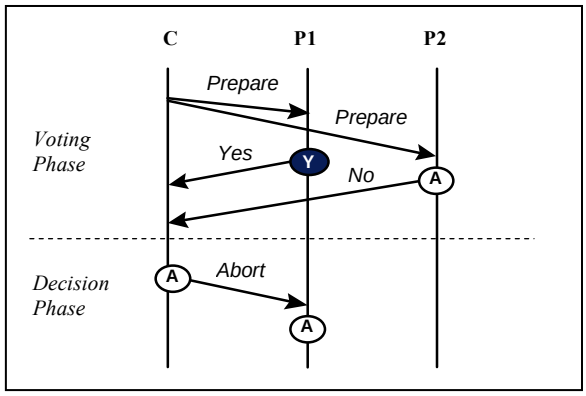

Figure 2.2: *The Presumed Abort protocol (Abort Case)*

2.4.1 Presumed Abort (PrA)

The basic 2PC protocol requires information to be explicitly exchanged and logged whether the transaction is to be committed or aborted. This is why it is often referred to as the *Presumed Nothing* 2PC (PrN) protocol. However, if after having failed and recovered, the coordinator of PrN gets an inquiry about the outcome of a transaction for which no information is found in its stable log, the coordinator (implicitly) presumes that the transaction is aborted.

The *Presumed Abort* optimization (PrA) [MLO86] exploits further this property in order to reduce the message and logging overhead associated with aborting transactions by making the implicit abort presumptions of PrN explicit. As illustrated in Figure 2.2, the coordinator of PrA does not log information nor wait for acknowledgments regarding aborted transactions. Consequently, participants do not acknowledge *abort* decisions nor log information about such decisions. To abort a transaction, the coordinator simply informs the participants of the *abort* decision and forgets about the transaction. In the absence of information about a transaction, the coordinator presumes that the transaction has been aborted. Regarding committing transactions, PrA behaves in exactly the same way as PrN.

It should be noted that PrA is now part of the ISO OSI-TP [ISO92a], X/Open DTP [X/Op91, X/Op93], and OMG OTS [OMG00a] distributed transaction processing standards, and has been implemented in a number of commercial products, such as IBM Almaden Research Center's R* [MLO86], Transarc's Encina [She93], and Unix System Laboratories' TUXEDO [Pri94].

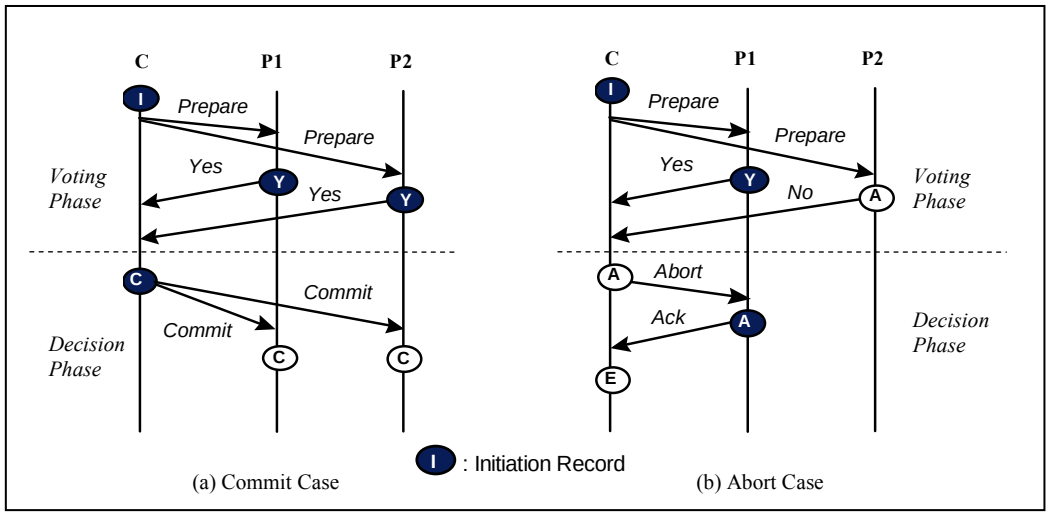

Figure 2.3: *The Presumed Commit protocol*

2.4.2 Presumed Commit (PrC)

The *Presumed Commit* protocol (PrC) [MLO86] is the counterpart of PrA in the sense that it reduces the cost associated with committing transactions. It is based on the observation that, in general, transactions are most likely to commit than to abort. In PrC, the coordinator interprets missing information about transactions as *commit* decisions.

Unlike PrA, however, the coordinator of PrC has to force-write a *membership* log record, and that, before starting the voting phase of the protocol. This is to ensure that an undecided transaction is not (erroneously) presumed as committed when the coordinator recovers from a crash. Furthermore, the *membership* record is exploited in order to record the identities of all the participants in the transaction, which, in the case of PrN and PrA, are recorded in the *decision* record.

As illustrated in Figure 2.3 (a), to commit a transaction, the coordinator of PrC force-writes a *commit* log record before sending the *commit* decision to the participants. This is actually needed so as to "logically" erase the *membership* record, since lack of information means a *commit*. When a participant receives the *commit* decision, it simply writes a non-forced *commit* record without acknowledging the decision. Figure 2.3 (b) illustrates the protocol behavior for aborting transactions. The coordinator writes a non-

forced *abort* record, and sends the *abort* decision only to those participants that voted *yes*. When a participant receives the *abort* decision from the coordinator, it force-writes an *abort* log record and then acknowledges the decision.

Although most transactions are expected to commit in the absence of failures, the argument usually goes in favor of PrA. Clearly, this is due to the extra logging activities associated with the *membership* record in PrC. Mechanisms for reducing the logging overhead of the original PrC and making its cost comparable to that of PrA have been proposed [ACL97, LaL93].

2.4.3 Decentralized 2PC (D2PC)

The *Decentralized 2PC* (D2PC) protocol [BHG87, Ske81] has been proposed in an attempt to reduce the time complexity of the basic 2PC. Instead of communicating through the coordinator, participants in D2PC communicate directly with one another. Similarly to PrN, the coordinator of D2PC initiates the protocol by sending a *prepare* message to all participants in the transaction. Unlike PrN, however, a participant that receives the *prepare* message responds by sending its vote to all participants in the transaction (rather than only to the coordinator). When a participant receives the votes from all participants, it decides on the transaction. If all votes are *yes* and the participant's own vote is *yes*, it decides *commit*; otherwise, it decides *abort*. Compared to the basic 2PC, D2PC eliminates one message round at the expense, however, of a quadratic message complexity assuming a point-to-point communication network ($n+2n^2$ messages, where n denotes the total number of participants in the transaction).

2.4.4 Read-Only

The Read-Only optimization [MLO86] has been proposed based on the observation that a transaction branch that performs only read operations cannot violate transaction atomicity. Since no local update has been performed on behalf of the transaction, a read-only participant does not care about the transaction outcome. When such a participant receives the *prepare* message, it sends a *read-only* vote (instead of a *yes* vote) and then immediately releases all the read locks it has acquired on behalf of its transaction branch.

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
PrN	$4n$	$2n + 2$	3	$2n + 1$
PrA	$4n$	$2n + 2$	3	$2n + 1$
PrC	$3n$	$n + 2$	3	$n + 2$
D2PC	$n + 2n^2$	$2n + 1$	2	$2n$

Figure 2.4 : *The cost of transaction commit under the different 2PC variations.*

The read-only vote has a dual role: it informs the coordinator that the transaction branch has read consistent data, and also tells it that the participant does not need to be involved in the second phase of the protocol. In short, a read-only participant does not perform any log write and sends only one message.

2.5 Performance Evaluation

The latency of an ACP is determined by the number of forced log writes and communication steps performed during the execution of the protocol, and until a decision is reached at every participant. Figure 2.4 shows the performances of 2PC together with its optimized variations in terms of latency and message complexity needed in order to commit a transaction (this actually corresponds to the most frequent case since most transactions are expected to commit in the absence of failures). Regarding message complexity, we distinguish two cases: (a) using a *point-to-point* network, and (b) using a *broadcast* network.

When compared to PrN (i.e., basic 2PC), PrA does not reduce the cost of committing transactions. Concerning PrC, we observe that although it reduces the number of messages and forced log writes, it does not reduce the number of communication steps required to commit a transaction. Furthermore, the force-writes that are saved at each participant in PrC are executed in parallel by the participants in PrN. Thus, PrC does not considerably reduce the overall latency of PrN given that it incurs an additional force-write associated with the *membership* log record at the coordinator site. D2PC reduces the time complexity of its centralized counterparts from three communication steps to two, which decreases the transaction response time.

Furthermore, D2PC requires one forced log-write less than the other 2PC variations due to the fact that, from the moment the coordinator of D2PC starts the protocol, it assumes exactly the same role as the other participants. As we have already pointed out, this gain in D2PC comes, however, at the expense of a quadratic number of messages exchanged during the protocol execution if a point-to-point communication network is used.

2.6 Discussion

In the light of the above study, it follows that, from a performance perspective, 2PC optimizations do not provide substantial benefits over basic 2PC. Thus, although adapted to the *classical* distributed environments and applications of their time, 2PC variations are far from being satisfactory when employed in *today's* highly reliable distributed platforms, and fail in meeting the strong performance requirements of *advanced* and *critical* applications, such as SCS applications [ABG98].

Beside this inefficiency, all 2PC variations require that the participating sites provide a local *prepared* state, which, as already mentioned in Chapter 1, violates site autonomy, precluding the integration of pre-existing legacy systems in distributed transactions [ShL90]. Although one might argue that this issue is no longer of topical interest as 2PC is now standardized, and hence *all* transactional systems are expected to become 2PC compliant, the actual situation shows that this is still not the case. Furthermore, and from a cost perspective, it would certainly be unreasonable to require that modifications be made to all existing transactional systems to support the standard protocol.

We believe that all these limitations constitute a strong argument towards a serious reconsideration of the two-phase commit approach, and explain the renewed interest in the atomic commitment problem.

Chapter 3

Dictatorial Atomic Commitment

The traditional transaction processing paradigm relies on a Two-Phase Commit approach to coordinate transaction termination. While Two-Phase Commit is indeed sufficient to guarantee transaction atomicity, one might wonder whether it is *always* necessary. This suggests that there might be room for a *One-Phase Commit* approach. In this chapter, we introduce the *Dictatorial Atomic Commitment* (DAC) problem, a novel paradigm for distributed transaction commit, which overcomes the need for Two-Phase Commit in most practical situations. The intuition behind Dictatorial Atomic Commitment is that the votes of the participants in a transaction introduce a high cost, and in most existing transactional systems, participants' votes can turn out to be more than necessary.

We first give a precise abstract specification of the Dictatorial Atomic Commitment problem, resulting from removing *veto rights* from the traditional Atomic Commitment problem, and describe a basic *One-Phase Commit* (1PC) algorithm that solves it. We then characterize transactional systems that are compatible with the DAC specification in terms of three necessary and sufficient conditions on concurrency control and recovery protocols. We also discuss the practical impacts of those conditions through an in-depth analysis of existing 1PC protocols. From this analysis, we point out some severe drawbacks related to the discussed protocols, which make them inapplicable to today's distributed systems. We finally propose a new 1PC variation that capitalizes on the existing ones so as to broaden the applicability of dictatorial transaction processing to meet the requirements of today's distributed environments and applications, and point out some interesting performance tradeoffs.

3.1 The Dictatorial Atomic Commitment Problem

3.1.1 Informal Description

Variations of 2PC solve the classical Atomic Commitment problem (specified in [BHG87]) by performing a voting phase and a decision phase. The possibility of a participant to vote *no* reflects its ability to reject a transaction a posteriori, i.e., after the transaction's operations are processed. In particular, a participant might need to vote *no* if it detects a risk of violating any of the local ACID properties of its transaction branch. Obviously, if we remove the *veto right* from participants in atomic commitment, the coordinator will not need to ask the participants for their votes and the *voting phase* of a 2PC becomes useless (cf. Figure 2.1).

Based on this idea, several authors have proposed the use of *One-Phase Commit* (1PC) protocols [StC90, StC93, AlC95, AlC96]. The basic assumption underlying 1PC is that a participant "does not need" to vote. This actually means that, before triggering the commit protocol, the coordinator of a 1PC makes sure that the ACID properties of all the local transaction branches are already ensured. In other words, the coordinator of a 1PC acts as a "nice dictator" and makes sure that no participant can have any reasonable reason to vote *no*. Obviously, this introduces some assumptions on the way participants manage their transactions as will be detailed later in the chapter.

3.1.2 Problem Definition

In light of the above discussion, we point out the fact that, by eliminating participants' votes, the problem solved by 1PC is no longer the classical Atomic Commitment problem solved by 2PC. This would contradict well-known lower bounds on the cost of solving atomic commitment in distributed transactional systems [DwS83]. In the following, we introduce the *Dictatorial Atomic Commitment* (DAC) problem, a distributed agreement problem resulting from removing veto rights from the traditional Atomic Commitment problem.

In Dictatorial Atomic Commitment, participants do not have the veto right. At commit time, the coordinator proposes one of two values: *commit* or *abort*. If the coordinator does

not crash, it forces the participants to accept its proposed value so that either they all commit the transaction or they all abort it. We formalize these notions as a set of properties that together define the Dictatorial Atomic Commitment problem [AGP00].

- **DAC-Uniform-Agreement:** No two participants reach different decisions.
- **DAC-Uniform-Validity:** The decision value is the coordinator's proposed value.
- **DAC-Uniform-Integrity:** No participant can reverse its decision after it has reached one.

The *DAC-Uniform-Validity* property clearly expresses the coordinator's dictatorship. The proposed value of the coordinator depends on whether or not the transaction has been successfully processed. A transaction is considered as successfully processed if all of its operations have been successfully executed and acknowledged by all participants. In this case, the coordinator proposes *commit*; otherwise, it proposes *abort*.

3.2 The Basic One-Phase Commit Protocol

In this section, we give a basic 1PC algorithm that solves the DAC problem, prove its correctness, and identify the different assumptions underlying it. An interesting feature of our algorithm is that it can be seen as the basic building block around which all existing 1PC variations are designed. Indeed, all 1PC protocols that were proposed in the literature share the same basic structure and differ only in the way recovery is managed (cf. Section 3.4.3).

3.2.1 Protocol Description

The simplest way to solve the DAC problem defined above is through the *terminate()* function described in Figure 3.1. During this function, the coordinator decides on the transaction depending on its *proposition* value, and sends its decision to all participants in the transaction. When a participant receives the decision from the coordinator, it decides on the transaction. Note that force-writing a decision record in the log is the act by which a participant decides on a transaction. The protocol corresponds exactly to a 2PC without the voting phase (see Figure 2.1). Clearly, one can apply various well-known optimizations of 2PC (e.g., PrA, or PrC) to 1PC.

```
function terminate ()

Only the coordinator executes:
1   decision: = proposition;                    // proposition ∈ {commit, abort}
2   decide (decision);
3   send (decision) to all other participants;
4   return;

Every participant P_i executes:
5   wait until [received (decision) from coordinator]
6         decide (decision);
7         return;
```

Figure 3.1: *The basic 1PC protocol*

3.2.2 Protocol Correctness

In this section, we prove the correctness of our basic 1PC algorithm presented in Figure 3.1. This amounts to proving that it satisfies all of the three properties of the DAC problem.

Theorem 3.1. *1PC achieves the DAC-Uniform-Agreement property.*

PROOF. For contradiction, assume that a participant P_i decides *commit*, while another participant P_k decides *abort*. In 1PC, a participant can only decide at line 6 following the receipt of the decision message from the coordinator (line 5). This means that the coordinator has sent two different decisions to participants P_i and P_k. This contradicts the fact that the coordinator sends the decision only once at line 3 of the protocol. Furthermore, it is clear that the decision sent by the coordinator at line 3 is nothing but the value it has decided at line 2. Thus, all participants (including the coordinator) reach the same decision.

Theorem 3.2. *1PC achieves the DAC-Uniform-Validity property.*

PROOF. From lines 1 and 2 of the protocol, it is obvious that the coordinator's decision value is its proposed value. By the *DAC-Uniform-Agreement* property, the decision value of all participants is the coordinator's proposed value.

Theorem 3.3. *1PC achieves the DAC-Uniform-Integrity property.*

PROOF. From the structure of the protocol, it is obvious that the coordinator decides at most once by executing line 2, while the other participants decide at most once by executing line 6.

3.2.3 Assumptions on the Transactional Systems

By interpreting acknowledgement messages as *yes* votes, the coordinator of 1PC verifies whether or not the ACID properties of the local transaction branches are already ensured at commit time. This obviously introduces some assumptions on the way participants manage their transactions. In the following, we give a precise identification of the different assumptions underlying 1PC, and usually made (explicitly or implicitly) by 1PC variations [AGP98]:

1. 1PC assumes that every transaction operation is acknowledged. Consequently, if the coordinator receives the acknowledgement messages for all the transaction operations before the protocol is launched, the *atomicity* of all the local transaction branches (i.e., local atomicity) will be already ensured at commit time.

2. 1PC assumes that integrity constraints are checked after each update operation and before acknowledging the operation. Thus, if all operations are acknowledged, *consistency* of all the local transaction branches will be already ensured at commit time (e.g., the possibility of discovering, at commit time, that there is not enough money for a bank account withdrawal is excluded).

3. 1PC assumes that a transaction that executes successfully all of its operations can no longer be aborted due to a serialization problem. Consequently, if all operations are acknowledged, *serializability* (isolation) of all the local branches will be already ensured at commit time (e.g., concurrency control protocols that check serializability at commit time are excluded).

4. Finally, 1PC assumes that once all operations are acknowledged, and before the protocol is launched, the effects of all the local transaction branches are already logged on stable storage, and hence, the *durability* property will be ensured at commit time.

We believe that assuming every operation to be acknowledged before the ACP is launched is not a strong requirement as most transactional standards like DTP from X/Open [X/Op93] and OTS from OMG [OMG00a] assume the same behavior. The second assumption means that deferred integrity constraints validation is excluded. However, the consequences of the last two assumptions are clearly less obvious. In the following two sections, we dissect these two assumptions and study their impact on the concurrency control and recovery protocols employed by participants in dictatorial atomic commitment.

3.3 The Impact of Dictatorship on Concurrency Control

In this section, we characterize schedulers that are correct without the need for a veto right at commit time. We give two necessary and sufficient correctness properties of such schedulers. The first property is an extension of *serializability*, which we named *on-line serializability* [AGP00], and the second is the well-known *cascadelessness* property [BHG87]. We show for instance that either strict *Two-Phase Locking* or strict *Timestamp Ordering* is sufficient to ensure *on-line serializability* and *cascadelessness*.

3.3.1 Veto Right Free Schedulers

The correctness of a scheduler is usually captured through two properties: *serializability* and *recoverability* [BHG87]. That is, a scheduler S is *correct* if only histories that are *serializable* and *recoverable* are acceptable for S. Roughly speaking, a scheduler does not need a veto right if it does not rely on a distributed voting phase to ensure either of these properties. For instance, the scheduler cannot optimistically authorize conflicts and decide to abort transactions at their termination time if the conflicts persist. In other words, an optimistic certifier does need a veto right. To capture these intuitive ideas, we first introduce the notion of *committed extension* of a history.

> **Definition 3.1.** Let H be any history. A *committed extension* of H is any history obtained by extending H with the commit operations of all active transactions in H.

Consider for example the following history[8]:

[8] In the notations, Ri[x] and Wi[x] denote respectively a Read (resp. Write) operation on object x performed by transaction Ti, while Ci and Ai denote the commit (resp. abort) of Ti.

$$H = W_1[x]\ R_1[y]\ W_2[z]\ A_1\ W_3[x]\ R_3[x]\ W_4[z]\ C_2$$

Both histories H1 and H2 below are *committed extensions* of H.

$$H1 = W_1[x]\ R_1[y]\ W_2[z]\ A_1\ W_3[x]\ R_3[x]\ W_4[z]\ C_2\ C_3\ C_4$$

$$H2 = W_1[x]\ R_1[y]\ W_2[z]\ A_1\ W_3[x]\ R_3[x]\ W_4[z]\ C_2\ C_4\ C_3$$

The following definition expresses the fact that a scheduler making use of 1PC (i.e., with no veto right at commit time) does not control the commitment of a transaction after its operations have been performed.

Definition 3.2. A scheduler S is *commit-expanded* if, whenever a history H is acceptable for S, any committed extension of H is also acceptable for S.

It is easy to see that a scheduler might be correct but not *commit expanded.* Let S be any correct scheduler (e.g., an optimistic certifier) for which the following history is acceptable:

$$H = W_1[x]\ R_2[y]\ W_2[x]\ R_1[x]$$

Now consider the following committed-extension of H:

$$H' = W_1[x]\ R_2[y]\ W_2[x]\ R_1[x]\ C_1\ C_2$$

The serialization graph of H' contains the cycle $T_1 \rightarrow T_2 \rightarrow T_1$, which means that H' is not *serializable*. The history H' is not recoverable either because transaction T_1 reads x from transaction T_2 and yet T_1 commits before T_2 ($C_1 < C_2$). As a consequence, H is acceptable for S whereas H' is not. In other words, S is not *commit-expanded*.

Definition 3.3. We say that a scheduler is *VR-free* (*veto right free*) if it is *correct* and *commit-expanded*.

3.3.2 On-line Serializability and Cascadelessness

The example above shows that *serializability* and *recoverability* are not sufficient for *VR-freedom*. In the following, we introduce a property, that we call *on-line serializability* [AGP00], which is stronger than *serializability*. Then we show that *on-line serializability* and *cascadelessness* (a history H is *cascadeless* if no transaction in H reads from values written by uncommitted transactions) [BHG87] are necessary and sufficient for *VR-freedom*.

To define *on-line serializability*, we introduce the *notation E-SG(H)* (*Expanded Serialization Graph*). Given a history H over a set of transactions T = {T_1, T_2, …T_n), *E-SG(H)* denotes the directed graph whose nodes are the transactions in T that are either committed or active in H and whose edges are all $T_i \rightarrow T_j$ (i ≠ j) such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. Note that *E-SG(H)* is a super-graph of *SG(H)* (the serialization graph of H) as the latest contains only committed transactions of H.

Definition 3.4. We say that a history H is *on-line serializable* iff *E-SG(H)* is acyclic.

Theorem 3.4. Let S be any *commit-expanded* scheduler. S is correct iff S ensures *on-line serializability* and *cascadelessness*.

PROOF.

(IF) Let S be any *commit-expanded* scheduler and assume that every history that is acceptable for S is *on-line serializable* and *cascadeless*. As for any history H, *E-SG(H)* is a super-graph of *SG(H)*, any cycle in *SG(H)* appears in *E-SG(H)* as well. Hence, any history that is not *serializable* is not *on-line serializable*. Furthermore, it was shown in [BHG87] that any history that is *cascadeless* is *recoverable*. Hence S is correct.

(ONLY IF) We show now that if a *commit-expanded* scheduler does not ensure either *on-line serializability* or *cascadelessness*, then it cannot be correct. Assume by contradiction that there is a history H in S that is either (1) not *on-line serializable* or (2) not *cascadeless*. Case (1) means that there is a cycle in *E-SG(H)*. Let H' be any *committed-extension* of H. As S is *commit-expanded*, then H' is acceptable for S. As *E-SG(H) = SG(H'),* then *SG(H')* also contains a cycle, a contradiction with the assumption that S is correct, i.e., S ensures *serializability*. Case (2) means that in H some transaction T_1 reads from values written by an uncommitted transaction T_2. Let H' be any *committed-extension* of H where T_1 commits before T_2. As S is *commit-expanded*, then H' is acceptable for S. Since H' contains all *read* and *write* operations of H, then in H', T_1 reads from values written by T_2, and T_1 commits before T_2 in H'. A contradiction with the fact that S is correct, i.e., S ensures *recoverability*.

Corollary 3.1. *On-line serializability* and *cascadelessness* are necessary and sufficient conditions for a scheduler to be *VR-free*.

3.3.3 Examples of VR-free Schedulers

We show below that a scheduler based either on strict *Two-Phase Locking* (2PL) or on strict *Timestamp Ordering* (TO) is *VR-free*.

Theorem 3.5. Strict 2PL is sufficient but not necessary to ensure *on-line serializability* and *cascadelessness*.

PROOF.

(a) It has been shown in [BHG87] that any strict history is *cascadeless*. Assume H is also a 2PL history and assume by contradiction that H is not *on-line serialisable*, i.e., there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in *E-SG(H)*. However, since 2PL is a lock-based scheduler, a dependency cycle would have led to a deadlock, and H could not have been generated: a contradiction.

(b) The following history H shows that strict 2PL is not necessary to ensure *on-line serialisability* and *cascadelessness*:

$$H = W_1[x]\ W_2[x]\ C_2\ C_1$$

The history H cannot be generated by a 2PL scheduler: transaction T_2 could not have accessed x before the termination of T_1. However, H is *on-line serializable* and *cascadeless*.

Theorem 3.6. Strict TO is sufficient but not necessary to ensure *on-line serializability* and *cascadelessness*.

PROOF.

(a) Similar to (a) of Theorem 3.5 above: assuming H is a TO history, the presence of a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in *E-SG(H)* would mean that ts(T1) < ts(T1), where ts(T) denotes T's timestamp. A contradiction.

(b) The following simple history H shows that strict TO is not necessary to ensure *on-line serializability* and *cascadelessness*

$H = W_1[x]\ W_2[x]\ C_1\ C_2$

Whatever the timestamp order is, H cannot be generated by a strict TO scheduler. Indeed, either ts(T_1) < ts(T_2) and W_2[x] will be delayed until C1 is performed, or ts(T_2) < ts(T_1) and T2 will be aborted because it arrives late. However, H is *on-line serialisable* and *cascadeless*.

In contrast, a certifier cannot ensure *on-line serializability*. A certifier typically prevents cycles by aborting transactions (a posteriori). However, *on-line serializability* requires that no cycle (even if involving only active transactions) be ever generated. The following history can be produced by a certifier and is obviously not *on-line serializable*.

$H = R_1[x]\ W_2[x]\ W_2[y]\ W_1[y]$

3.3.4 Practical Considerations

Strict 2PL is the most widely used serialization protocol. Hence, participants of most transactional systems exhibit the *VR-free* property and thus, are 1PC compliant. However, commercial database systems are likely to use *isolation levels* standardized by SQL2 [ISO92b] in combination with 2PL. We recall below the SQL2 isolation levels and analyze the extent to which 1PC protocols can accommodate them.

- *Serializable*: Transactions running at this level are fully isolated.
- *Repeatable Read*: Transactions running at this level are no longer protected against phantoms. More precisely, successive reads of the same object give always the same result but successive SQL queries selecting a group of objects may give different results if concurrent insertions occur.
- *Read Committed*: Transactions running at this level read only committed data but *Repeatable Read* is no longer guaranteed. In a lock-based protocol, this means that read locks are relaxed before transaction end (in practice, as soon as they are granted).
- *Read Uncommitted*: Transactions running at this level may do dirty reads. For this reason, they are not allowed to update the database. In a lock-based protocol, this means that *Read Uncommitted* transactions do not request locks at all.

Isolation levels are widely exploited because they allow faster executions, increase transaction parallelism and reduce the risk of deadlocks. For example, a transaction T_i computing statistics on a large population of objects can take benefit of the *Read Uncommitted* level. This transaction will never be blocked by concurrent writing transactions (that may affect T_i's result but in a non significant way) and will never block other transactions.

If we refer to definition 3.3, it is clear that schedulers implementing isolation levels, which we call *IL-schedulers*, are not *VR-free* simply because they are not correct: they do not ensure *serializability*. Consequently, they do not ensure *on-line serializability* either. However, isolation levels have been actually introduced to relax serializability, and non-serializable schedules that may be produced are considered as semantically correct. Hence, new correctness criteria that accommodate isolation levels need to be defined in order to characterize "correct" *IL-schedulers*. To this end, we introduce in the following a new property, which we call *IL-serializability*.

Consider a history H over a set of transactions T = {T_1, T_2, …, T_n). Let *IL-SG(H)* be the sub-graph of *SG(H)* containing all dependencies in H except those incurred by conflicts ignored by the isolation levels under which transactions in T are running. We say that H is *IL-serializable* iff *IL-SG(H)* is acyclic. An *IL-scheduler* is said to be *correct* if it ensures *IL-serializability* and *recoverability*.

Similarly to Section 3.3.2, we introduce a new property, which we call *on-line IL-serializability,* to characterize *IL-schedulers* that are correct with no veto right at commit time. Let *E-IL-SG(H)* denote the expanded *IL-SG(H)*. We say that a history H is *on-line IL-serializable* iff *E-IL-SG(H)* is acyclic. We can show that *on-line IL-serializability* and *cascadelessness* are necessary and sufficient conditions for an *IL-scheduler* to be *veto right free*. The proof is very similar to that of Theorem 3.4 and hence omitted. We show below that IL-2PL (2PL based IL-scheduler) satisfies both *cascadelessness* and *on-line IL-serializability*.

- ***Cascadelessness:*** conventionally, the *cascadelessness* property precludes the occurrence of dirty reads. In *IL-2PL*, dirty reads are allowed only at the *Read Uncommitted* level, which is restricted to *Read-Only* transactions. However, the semantics of *Read-Only* transactions contradict the fact that they can be subject to

cascading aborts. Consequently, *cascadelessness* is still ensured in *IL-2PL* schedulers.

- ***On-line IL-serialisability:*** Assume H is an *IL-2PL* history, and assume by contradiction that H is not *on-line IL-serialisable*, i.e., there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in *E-IL-SG(H)*. Note that any dependency edge in *E-IL-SG(H)* translates a conflict not ignored by the *IL-2PL* scheduler. Since *IL-2PL* is based on locking, a dependency cycle would have led to a deadlock and H could not have been generated: a contradiction.

As a conclusion, *IL-2PL* schedulers can still be considered as *veto right free*, and hence they comply with 1PC.

3.4 The Impact of Dictatorship on Recovery

A data manager must ensure the *atomicity* and *durability* properties of every transaction. More precisely, the data manager must guarantee that there is enough information on stable storage so that if a failure occurs (and the information in the volatile storage is lost), (1) the updates of aborted transactions are undone from the database and (2) the updates of committed transactions are correctly reported on the database. Following the terminology of [Had88], we call the first property *abort-resiliency* and the second property *commit-resiliency* (these correspond to *undo* and *redo* rules respectively in [BHG87]). A data manager is said to be *correct* if it guarantees both *abort-resiliency* and *commit-resiliency* [Had88].

3.4.1 Veto Right Free Data Managers

In a centralized system, *abort-resiliency* is for example ensured by having the data manager store *before images* in its log (this technique relies on the assumption that a strict concurrency control is used), and *commit-resiliency* is ensured by force-writing the transaction updates on stable storage at commit time [BHG87].

In a distributed database system, the same technique is used to guarantee *abort-resiliency*. To ensure *commit-resiliency*, participants in a transaction must guarantee that,

if the transaction commits at any participant, there is enough information on stable storage to *redo* the effects of the transaction at *all* participants. With a 2PC, this is guaranteed using the notion of *prepared* state. A participant *P* enters the *prepared* state for a transaction only if the *commit-resiliency* property is guaranteed for the transaction branch that accessed *P*. To commit a transaction, its coordinator makes sure that all updated participants have entered the *prepared* state of that transaction: this test is included in the *voting* phase of the 2PC. A participant does only vote *yes* if it has entered the *prepared* state. If it cannot enter that state (e.g., if the disk is full), the participant simply votes *no* and the transaction is aborted.

Removing the veto right has no impact on *abort-resiliency*. Nevertheless, the participants must anticipate the commit and make sure *the commit-resiliency* property is ensured a priori. As for schedulers, we introduce the following definitions to capture the idea of a *VR-free* data manager [AGP00].

> **Definition 3.5.** We say that a data manager D is *commit-expanded* if whenever an operation has been performed on behalf of a transaction T, the corresponding transaction branch can commit.

The definition above captures the idea that, just like for a scheduler, the only way to abort a transaction is by not performing one of its operations. If a transaction's operation has been acknowledged (i.e., performed), the corresponding transaction branch is able to commit.

> **Definition 3.6.** We say that a data manager is *VR-free* if it is *correct* and *commit-expanded.*

3.4.2 On-line Commit-resiliency

We introduce the following property to characterize the behavior of data managers that are *VR-free*.

> **Definition 3.7.** We say that a data manager ensures *on-line commit-resiliency* if every *update* operation executed on that data manager is *commit-resilient*.

Theorem 3.7. Let D be any *commit-expanded* data manager. D is *correct* iff it ensures *abort-resiliency* and *on-line commit-resiliency.*

PROOF.

(IF) Let D be any *commit-expanded* data manager that ensures *abort-resiliency* and *on-line commit-resiliency*. In other words, before acknowledging any update operation, the participant force-writes its effects on stable storage. As we assume that this participant cannot commit its transaction branch before all of its operations have been acknowledged (cf. Sections 3.1 and 3.2), this means that it cannot commit its transaction branch if the effects of any of its operations are not on stable storage, i.e., the transaction is *commit-resilient* at D's site. Hence, D is correct.

(ONLY IF) Assume by contradiction that there is an execution where D does not ensure *on-line commit-resiliency*, i.e., D does not ensure the *commit-resiliency* of some update operation *op* for a transaction *T*. If the transaction commits exactly after receiving the acknowledgement from the participant about the operation *op*, and the participant crashes immediately after sending back that acknowledgment, then the effects of *op* are lost and *T* is not *commit-resilient* at D's site: a contradiction with the fact that D is correct.

Corollary 3.2. *Abort-resiliency* and *on-line commit-resiliency* are necessary and sufficient conditions for a data manager to be *VR-free*.

3.4.3 Practical Considerations

Participant Logging

To achieve the *on-line commit-resiliency* property, participants in a transaction must force-write the effects of every update operation on stable storage, and that before acknowledging the operation. The *Early Prepare* (EP) processing scheme of Stamos and Cristian does ensure that property [StC90, StC93]. Although Early Prepare can make direct use of 1PC (as described in Section 3.2.1) and alleviates the need for an expensive 2PC, it requires a forced-write at every update operation of the transaction. The cost of transaction commitment is hence traded with the cost of transaction processing.

Coordinator Physical Logging

To avoid the prohibitive cost of *on-line commit-resiliency*, one might deviate from the "classical" atomic commitment scheme that requires every participant to ensure all of the ACID properties of its transaction branches. Consider for instance a less classical scheme that consists in having the coordinator itself ensure the *commit-resiliency* property before committing a transaction. To delegate this responsibility, participants need however to make sure that the coordinator has enough information on its local stable log about all committed transactions (unless it has the adequate information, the coordinator aborts the transaction). *Coordinator Log* (CL) [StC90, StC93] and *Implicit Yes-Vote* (IYV) [AlC95, AlC96] do follow this scheme.

In Coordinator Log, participants do not maintain their updates in a local stable log. Instead, they send back within the acknowledgment message of every update operation *all the log records* (undo and redo log records) generated during the execution of the operation. The coordinator is thus in charge of logging the transaction's update information before performing the commit protocol. If we refer to the basic 1PC protocol described in Section 3.2.1, this would mean that the coordinator of CL calls the *terminate()* function with *commit* as its proposition value only if it succeeds in storing the transaction's log records on stable storage. To recover from a crash, a participant asks the coordinator for the undo/redo log records it needs to reestablish a consistent state of its database.

The Implicit Yes-Vote scheme is similar to Coordinator Log, except that logging is a more distributed task. The idea is to allow failed participants to perform part of the recovery procedure (the undo phase) independently of the coordinator, and to resume the execution of transactions that are still active in the system (i.e., transactions for which no decision was made yet) instead of aborting them. Participants send back their redo log records together with a *Log Sequence Number* (LSN) [GrR93] whenever they acknowledge an update operation. To recover from a crash, a participant performs the undo phase of the recovery procedure and part of the redo phase using its local log. Then, the participant asks the coordinator for all redo log records whose LSNs are greater than its own highest LSN, and for all read locks acquired by active transactions. This allows the participant to reinstall the updates pertaining to globally committed transactions and continue the execution of transactions that are still active in the system.

Coordinator Logical Logging

Although Coordinator Log and Implicit Yes-Vote circumvent the need for *on-line commit-resiliency*, they violate site autonomy by forcing participants in a transaction to externalize their local log information. This certainly compromises their use in existing transactional systems. To solve this problem, we propose to maintain in the log of the coordinator the list of operations submitted to each participant instead of the physical redo log records sent back by these participants. In case a participant crashes during the 1PC protocol execution, the failed transaction branches that make part of a globally committed transaction will be re-executed using the operations registered in the coordinator's log.

This mechanism, which we call *Coordinator Logical Log* (CLL) [AbP98], provides three main advantages. First, it preserves site autonomy since no internal information has to be externalized by the participants. This feature is of primary importance if the commit protocol is to be used in today's large and autonomous distributed environments. Second, it can be applied to heterogeneous transactional systems using different local recovery schemes, which is not the case in Coordinator Log or Implicit Yes-Vote. Finally, it does not increase the communication cost during normal processing since log records are not piggybacked in the messages.

3.5 The CLL Protocol

3.5.1 Failure-Free Execution

As introduced before, our logical logging mechanism consists in having the coordinator register in its log every transaction operation before sending it to the participant hosting the object involved by the operation. Note that this registration is done by a non-forced write. Non-forced writes are buffered in main memory and do not generate blocking I/O. Operations are then sent to and locally executed by the different participants.

As is the case in CL and IYV, the coordinator of CLL is in charge of ensuring the *commit-resiliency* property before committing a transaction. Thus, when all acknowledgments are received, the coordinator force-writes the transaction operations on

stable storage and calls the *terminate()* function with *commit* as its proposition value. Recall that during this function, the coordinator decides on the transaction by force-writing its decision value on disk. In order to improve performances, the transaction operations together with the decision log record can be forced on stable storage at the same time, thereby generating a single blocking I/O. If, on the other hand, the coordinator receives a negative acknowledgement from some participant or fails in storing the transaction operations on stable storage, it simply discards all the transaction's log records and calls the *terminate()* function by proposing *abort*.

3.5.2 Dealing with Failures

Similarly to the AC problem, the following *DAC-Termination* property has to be added to the specification of the DAC problem in order to exclude protocols that allow participants to remain undecided forever once some failures have occurred during the protocol execution.

- **DAC-Termination:** If all failures are repaired, then unless a new failure occurs, every participant eventually reaches a decision.

To satisfy DAC-Termination, we must supply timeout actions for each point in the 1PC protocol in which a participant is waiting for a message. Timeout actions define what a participant should do in case an expected message does not arrive. We must also describe how a recovering participant attempts to reach a decision consistent with the decision other participants may have reached in the meanwhile. In the following, we consider these two issues in turn[9].

Timeout Actions

In 1PC, the only point where a participant can unilaterally abort a transaction is by negatively acknowledging an operation. If, however, the participant has no pending acknowledgement for any of the transaction's operations, it enters its uncertainty period until it receives either a new operation or the final decision from the coordinator. When a participant times out while in its uncertainty period, it executes a termination protocol

[9] For details on how crash recovery and timeout actions are handled in CL and IYV, please refer to [StC90, StC93, AlC95, AlC96].

during which it tries to decide on the transaction. The termination protocol presented in Section 2.3.2 can be perfectly used here so that *DAC-Termination* is guaranteed. Note that although the participant may be blocked during the execution of the termination protocol due to failures in other parts of the system, it eventually reaches a consistent decision once these failures are repaired.

Crash Recovery

We now describe how a recovering participant can reach a decision consistent with the decision operational processes may have reached. Consider a participant P_k recovering from a crash. Figure 3.2 details the recovery algorithm associated with CLL and executed by P_k. In the following, T_{ik} denotes the local branch of transaction T_i executed at participant P_k. For the sake of clarity, step numbers correspond here to step ordering.

Step 1 and Step 2 represent the standard local recovery procedure executed by a crashed participant P_k. To preserve site autonomy, we make no assumptions whatsoever on the way these steps are handled. Step 3 is necessary to determine if the k^{th} branch (i.e., T_{ik}) of some globally committed transactions T_i has to be locally re-executed by the crashed participant. In Step 4, the coordinator aborts all active transactions in which P_k participates.

Step 5 checks if there exists some committed transaction T_i for which P_k did not acknowledge the *commit* decision. This may happen in two situations. Either the participant crashed before the commit of T_{ik} was achieved and T_{ik} has been undone during Step 1, or T_{ik} is locally committed but the crash occurred before the acknowledgment was sent to the coordinator. Note that these two situations must be carefully differentiated. Re-executing T_{ik} in the latter case may lead to inconsistencies if T_{ik} contains *non-idempotent* operations. To simplify the presentation, we assume for the moment that the coordinator can query a participant to learn the exact state of T_{ik} (Step 6). We detail afterwards the way we achieve this without violating site autonomy. The participant answers during Step 7. If T_{ik} has been successfully committed, the coordinator does nothing. Otherwise, T_{ik} has been undone during Step 1 and must be entirely re-executed. This re-execution is performed by exploiting the coordinator's log (Step 8). Once the recovery procedure is completed, new distributed transactions are accepted by the coordinator (Step 9) and the participant (Step 10).

Participant's algorithm

1- *undo the transactions that were active at the time the crash occurred*

2- *redo transactions that have reached their commit state before the crash occurred*

3- *contact the coordinator*

7- *answer the queries that may be sent by the coordinator during step 6*

10- *accept new transactions*

Coordinator's algorithm

If contacted by participant P_k during step 3, do:
for each transaction T_i in which P_k participates

4- *if ($commit_i$) $\notin$ coordinator's log, then*
send ($abort_i$) to all other T_i participants and forget T_i

5- *if ($commit_i$) $\in$ coordinator's log, then*
if $ack_k(commit_i) \notin$ coordinator's log, then

6- *query P_k to determine the exact status of T_{ik}*
(i.e., either locally committed or aborted)

8- *if T_{ik} has not been locally committed, then*
restart a new transaction T'_{ik} on P_k
re-execute all T_{ik} operations within T'_{ik}

9- *accept new distributed transactions*

Figure 3.2. *Recovering a participant crash*

We now explain how the coordinator can query a participant about the state of its local transaction branches. Our solution relies on a local *Agent* (called $Agent_k$) associated with each participant P_k. The *Agent* does not violate site autonomy as the existing interface of the participant is preserved, and does not increase the communication cost, as it is co-located with its participant. Every message is submitted to the participant through its local *Agent*, which acts as a liaison between the coordinator and the participant. The exact role of the *Agent* is to determine, during the recovery procedure, those local transaction branches that need to be re-executed. The mechanism works as follows. When the coordinator sends the *commit* decision to each participant, the participant's *Agent* issues an additional operation "write record <$commit_i$>" on behalf of the local transaction branch it is in charge of (e.g., T_{ik}), and before submitting the *commit*

decision to the participant [10]. This creates at P_k a special local record containing the *commit* decision for T_i. This operation will be treated by P_k in exactly the same manner as the other operations belonging to T_{ik}, that is, either all committed or all aborted atomically. Once the *Agent* receives the acknowledgment of this write operation, it asks P_k to commit the local transaction branch.

Steps 6 and 7 of the recovery algorithm are now straightforward. To get the status of a local transaction branch T_{ik}, the coordinator checks, through $Agent_k$, the existence of record <commit$_i$> at P_k (this can be done by a regular select operation). If the record is found, this proves that T_{ik} has been successfully committed at P_k before the crash, since write<commit$_i$> is performed on behalf of T_{ik}. Otherwise, T_{ik} has been backward recovered during Step 2 and must be re-executed.

3.5.3 Recovery Correctness

In this section, we show that the CLL's recovery procedure described in the previous section is correct. This amounts to proving that a recovering participant eventually reaches a decision consistent with that reached by the other participants once all failures are repaired so that *DAC-Termination* is satisfied. However, since the recovery procedure may lead to a decision through the re-execution of a transaction branch, we also need to show that re-executing the logical operations registered in the coordinator's log will produce exactly the same local state at the recovering participant as the one produced during the initial execution. In the following, we consider these two issues in turn.

- ***Decision Consistency:*** Let P_k be the recovering participant. If, during its local recovery procedure, P_k finds in its log a decision record for a transaction branch, say T_{ik}, then it has already decided during the 1PC protocol execution. If, however, no decision record is found, P_k undoes the effects of T_{ik} (Step 1). Note that the only non-trivial case to consider here is the case where T_{ik} is part of a globally committed transaction T_i. This may happen if the coordinator has sent the commit decision to all participants, but P_k crashed before committing T_{ik}. By the algorithm of Figure 3.2, when P_k establishes a

[10] Note that this operation never generates a dependency cycle (i.e., deadlock) since it is the last operation executed in any transaction that has to be committed.

consistent local state, it contacts the coordinator (Step 3). In this case, once the coordinator has verified, through $Agent_k$, that T_{ik} has been locally aborted, it re-executes all T_{ik} operations within a new transaction branch T'_{ik}. If a failure should occur during the re-execution process, it will be retried until T_{ik} (T'_{ik}) commits at P_k. Note that although P_k may be blocked during its recovery (in case the coordinator is down), P_k eventually reaches a consistent decision once the coordinator recovers from its crash. Hence, the recovery procedure associated with CLL satisfies *DAC-Termination*.

- ***Determinism:*** Here, we show that the re-execution of T_{ik} within T'_{ik} produces the same local state at P_k as the one produced during the initial execution. Note that in CL and IYV, the coordinator's log contains physical redo records, making the recovery algorithm rather straightforward. The redo records are re-installed at the failed participant during the recovery of a local transaction branch, thereby producing the same local state as the one produced during the initial execution. By exploiting logical logging rather than physical logging, CLL's recovery procedure must face two new problems:

  - operations may be non-idempotent: an operation *op* is said to be non-idempotent if (op(op(x)) ≠ op(x)). Non-idempotent operations must be executed exactly once in any failure situation.

  - operations may be non-commutative: two operations *op1* and *op2* are said to be non-commutative if (op1(op2(x)) ≠ op2(op1(x))). Non-commutative operations must be executed at recovery time in the same order as during the initial execution.

Consider first the management of non-idempotent operations. Assume the coordinator has decided to commit T_i and has sent its decision to the participants. Assume also that P_k crashed immediately after. By the *undo* rule, if P_k crashed before committing T_{ik}, T_{ik} will be undone during Step 1 of the recovery algorithm and the record <commit$_i$> will be discarded[11]. Otherwise (i.e., P_k crashed after the commit of T_{ik} was successfully performed), the *redo* rule guarantees the presence of the <commit$_i$> record at P_k. These two situations are differentiated during Step 6 of the recovery algorithm. Step 8 forward recovers only transaction branches that have been locally aborted. This means that no transaction branch, and hence no operation (whether idempotent or not) is executed twice.

[11] We recall that the operation *write record* <$commit_i$> is performed on behalf of T_{ik}.

Consider now non-commutative operations. If these operations belong to the same transaction, no problem can occur. Indeed, the recovery algorithm re-executes the operations of a failed transaction branch following the order in which they were logged on the coordinator, i.e., in the order of their initial execution. The case where two or more local transaction branches (e.g., T_{ik}, T_{jk}) have to be forward recovered is trickier since most transactional systems execute transactions in parallel through several threads of control. Thus, even if the coordinator re-submits to P_k all operations that belong to different local transaction branches in the order of their initial execution, the result is non-deterministic. We demonstrate below that the local database state produced by the recovery algorithm is the same as the one produced during the initial execution. Let φ denote the set of all local transaction branches that have to be forward recovered by P_k during Step 8.

$$\varphi = \{T_{ik} \,/\, \text{commit}_i \in \text{coordinator's log} \wedge \text{ack}_k(\text{commit}_i) \notin \text{coordinator's log} \wedge \langle \text{commit}_i \rangle \notin P_k\text{'s state}\}$$

First, Step 2 of the recovery algorithm guarantees that all resources accessed by any $T_{ik} \in \varphi$ are restored to their initial state (i.e., the state before T_{ik} execution), according to the *atomicity* property. Second, since Step 8 precedes Step 9 and Step 10, new transactions that may modify T_{ik} resources are executed only after the re-execution of T_{ik}. Consequently, at Step 8, all $T_{ik} \in \varphi$ are guaranteed to re-access the initial database state. The sole problem may come from the parallel re-execution of all $T_{ik} \in \varphi$ if these transactions themselves compete on the same resources.

Assume first that P_k uses a locking based *VR-free* serialization protocol, such as strict *2PL* (i.e., the general case). In this case, $\forall T_{ik}, T_{jk} \in \varphi, \neg\exists(T_{jk} \rightarrow T_{ik})$, where $\rightarrow$ represents a precedence in the serialization order. Otherwise, T_{ik} would have been blocked during its initial execution, waiting for the termination of T_{jk}, and would not have completed all its operations, which contradicts $T_{ik} \in \varphi$. This means that T_{ik} and T_{jk} cannot compete on the same resources. If however, P_k uses another *VR-free* serialization protocol, such as strict *TO*, the former assumption is no longer valid. Indeed, strict *TO* accepts some Read/Write conflicts (those produced in the timestamp order) without blocking. To deal with this case, Step 8 must execute each $T_{ik} \in \varphi$ in their initial serialization order, one after the other (i.e., without parallelism).

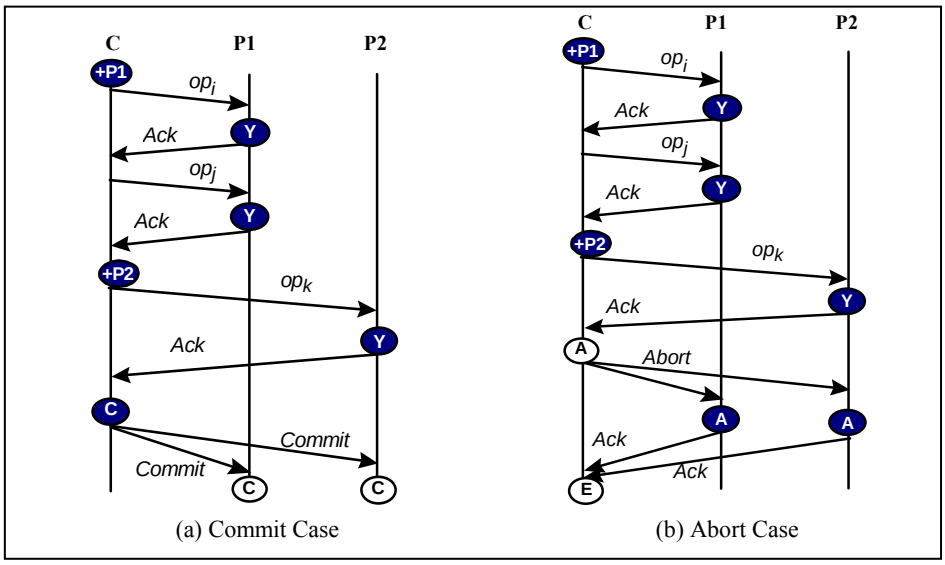

Figure 3.3: *The EP protocol.*

3.5.4 Performance Evaluation

In this section, we investigate the performance of One-Phase Commit, and compare the CLL protocol with existing 1PC variations, namely EP, CL and IYV. In our evaluations, we denote by n the total number of participants in a transaction, and we assume failure-free executions.

As opposed to the basic 2PC, which requires 3 communication steps, $2n$+1 log forces, and $4n$ messages in order to commit/abort a transaction (cf. Section 2.5), the basic 1PC protocol (described in Section 3.2) only requires one communication step, n+1 log forces, and $2n$ messages. The absence of the veto right explains why 1PC is much more efficient than 2PC.

While basic 1PC treats all transactions uniformly, whether they are to be committed or aborted, one can clearly apply various well-known optimizations of 2PC (e.g., PrA and PrC) to 1PC. The EP protocol combines the 1PC idea with PrC in the sense that it reduces the message and logging overheads associated with committing transactions. Consequently, commit decisions are neither acknowledged nor force-written by the participants. However, since the coordinator of PrC must record the identities of the transaction participants on stable storage as part of a forced *membership* log record, and that, before sending prepare messages to the participants (cf. Section 2.4.2), the

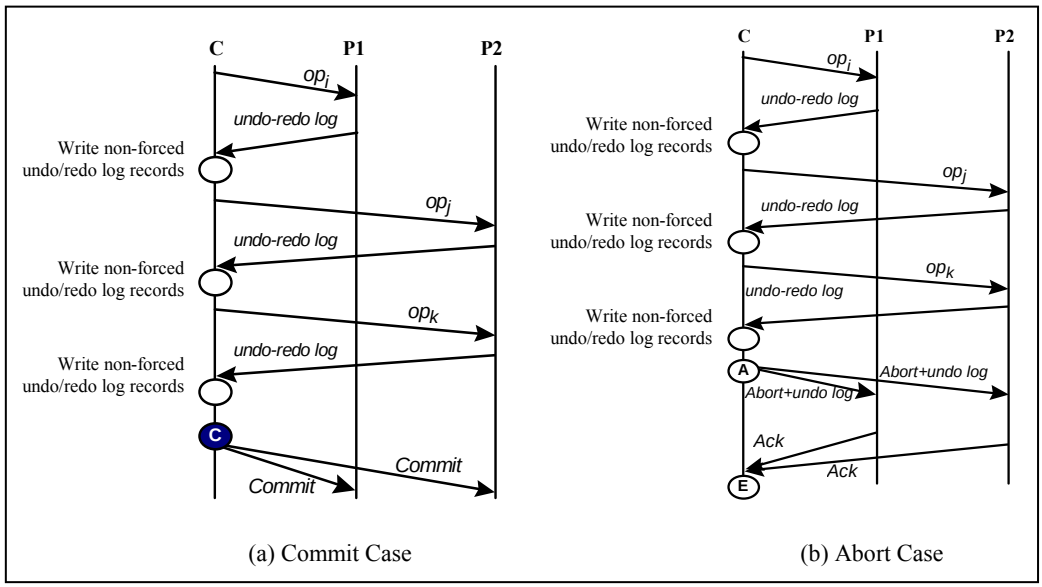

Figure 3.4: *The CL protocol.*

coordinator of EP may have to force-write multiple *membership* log records, because the transaction membership may grow as the transaction execution progresses. Furthermore, by achieving *on-line commit resiliency* (cf. Section 3.4.3), EP generates one force-write for each update operation. This makes a total of $1+n+op$ log forces for the commit case, and $2n+op$ log forces for the abort case, where *op* denotes the number of update operations performed by a transaction. Figure 3.3 illustrates the protocol behavior for committing as well as aborting transactions.

Another 1PC variation that is based on PrC is the CL protocol (Figure 3.4). Unlike EP, however, CL eliminates the forced *membership* log record of PrC by requiring a recovering coordinator to communicate with every possible participant in the system in order to determine all the transactions that were active at the time of the crash, and to abort them instead of wrongly presuming commit. This means, however, that coordinators in CL cannot independently recover, and must wait for all participants in the system in order to resume execution. Furthermore, as discussed in Section 3.4.3, CL overcomes the high cost of *on-line commit resiliency* by implementing *distributed write-ahead logging* (DWAL) in order to give up any logging activity at the participants. The combination of this mechanism with PrC results in a severe problem since transactions' updates must be remembered forever, and hence, the coordinator's log can never be garbage collected!

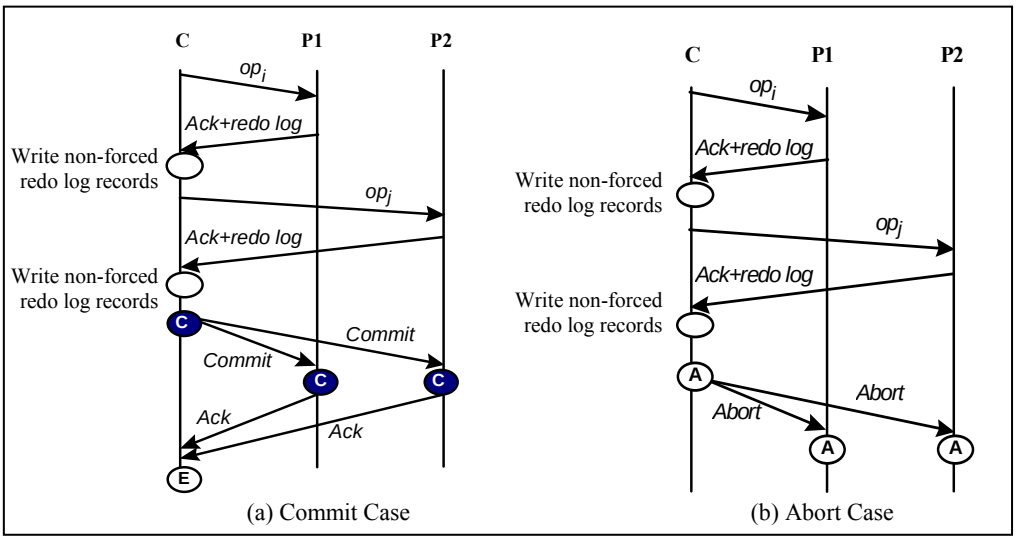

Figure 3.5: *The IYV protocol.*

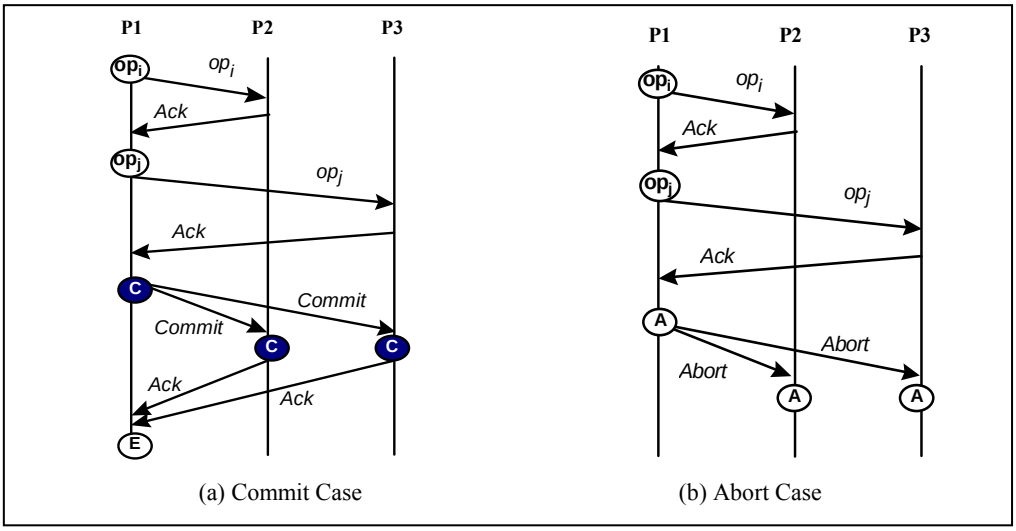

Figure 3.6: *The CLL protocol.*

From the above discussion, it follows that the implications of the combination of 1PC with PrC are severe. Consequently, unlike EP and CL, the IYV protocol (Figure 3.5) and the CLL protocol (Figure 3.6) exploit the PrA optimization by adopting abort presumptions. Thus, for committing transactions, both protocols have the same performances as basic 1PC, and reduce the message and logging overheads for aborting ones. Similarly to CL, IYV and CLL eliminate the high logging cost of *on-line commit resiliency*. In IYV, however, this is achieved by implementing a *replicated write-ahead logging* mechanism (RWAL), whereas in CLL by implementing a *coordinator logical logging* mechanism.

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
EP	n	1	1	$n + op + 1$
CL	n	1	1	1
IYV	$2n$	$n + 1$	1	$n + 1$
CLL	$2n$	$n + 1$	1	$n + 1$

Figure 3.7: *The cost of transaction commit under the different 1PC variations.*

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
EP	$2n$	$n + 1$	1	$2n + op$
CL	$2n$	$n + 1$	1	0
IYV	n	1	1	0
CLL	n	1	1	0

Figure 3.8: *The cost of transaction abort under the different 1PC variations.*

Figure 3.7 and Figure 3.8 summarize the cost of the different 1PC variations in terms of latency and message complexity needed in order to commit and abort a transaction, respectively. Regarding message complexity, we distinguish two cases: (a) using a *point-to-point* network, and (b) using a *broadcast* network. We recall that n denotes the total number of participants in the transaction, while op stands for the number of update operations performed by a transaction.

For the commit as well as the abort case, the cost associated with EP is highly dependent on the number of participants in a transaction and on the number of update operations performed by the transaction. This makes EP quite inefficient when used in today's large distributed systems, where transactions are most likely to span several sites, and to execute an important number of operations at these sites. Thus, unless every transaction performs at most one update operation at every site, or the sites are equipped with electronic stable storage (i.e., free log forces), the cost of EP can turn out to be far more prohibitive than the cost of a 2PC.

By combining PrC with DWAL, CL outperforms the other variations in the commit case as far as log forces are concerned, and shares with EP the lowest message complexity. However, as already stated before, the price of this efficiency is a set of severe drawbacks resulting, on one hand, from a coordinator's log that can never be garbage collected (a rather unrealistic assumption), and on the other hand, from a coordinator recovery procedure that totally depends on every possible participant in the system. Furthermore, participants in CL cannot locally handle aborted transactions, not to mention unilateral aborts! This is because the undo records are maintained only at the coordinator site. Hence, undoing a transaction has to be completely performed over the network, and local resources held by an aborted transaction cannot be released by a participant before getting the necessary undo records from the coordinator. This leads to a quick degradation in CL's performances, making it much more suitable for parallel architectures rather than geographically distributed systems.

For the abort case, IYV and CLL have the best overall performances, and share with CL the lowest logging overhead. Even though, by combining RWAL with PrA, IYV overcomes the abovementioned problems introduced by CL, both protocols require that participants in a transaction externalize their local log information. This means that major modifications should be made to existing transactional systems in order to support CL or IYV, which is definitely unacceptable in today's large distributed environments in which local site autonomy is of key importance. By combining a logical logging mechanism with PrA, CLL capitalizes on both CL and IYV. This leads to the conclusion that, among all the discussed protocols, CLL offers the best tradeoff between performance and compliance with existing transactional systems. Therefore, it appears to be the best candidate for distributed transaction termination in today's distributed environments and applications.

3.6 Discussion

One-Phase Commit is a highly efficient approach to distributed transaction commit that is based on a Dictatorial Atomic Commitment paradigm. The intuition behind 1PC is that veto rights in the traditional 2PC introduce a high cost, and this cost should only be paid when necessary.

The advantages of 1PC over 2PC are not only performance issues. By eliminating participants' votes, 1PC overcomes the various problems incurred by the local prepared state required in 2PC. Obviously, the appealing features of 1PC have a price, which we expressed in terms of three necessary and sufficient conditions on concurrency control and recovery protocols employed by the participants in a transaction. When adequately exploited, however, we have shown that 1PC offers a highly efficient approach to distributed transaction commit that is applicable to most practical situations. In particular, we proposed a new 1PC protocol that exploits a Coordinator Logical Logging mechanism in order to achieve correct recovery without compromising site autonomy, making it the sole protocol that can cope with all existing transactional systems — be they or not 2PC compliant.

Part II

Fault-Tolerance Issues

Chapter 4

Non-Blocking Atomic Commitment: Background

Although the atomic commitment protocols we have discussed thus far guarantee transaction atomicity, which is a safety condition, they do not provide liveness guarantees, i.e., they may lead to *blocking* situations in which participants are unable to decide on the transaction due to failures in other parts in the system. Consequently, a transaction can hold valuable system resources for an unbounded period, making these unavailable to other transactions that in turn become blocked, which may eventually block the entire system.

The impact of indefinite blocking and long-duration delays is particularly aggravated in mission critical applications (e.g., SCS) or applications involving an important number of sites (e.g., Internet). Furthermore, in today's large distributed environments in which the various sites participating in the system may belong to several *autonomous*, and possibly *competing* business organizations, it would be unconceivable to allow a remote transaction belonging to a competing organization from blocking local resources. An atomic commitment protocol is said to be *non-blocking* if it allows a decision to be reached at every correct participant despite failures of others.

This chapter gives some background about the *Non-Blocking Atomic Commitment* (NB-AC) problem, and presents a survey of non-blocking commit protocols commonly found in the literature. In order to do so, we refine the general system model described in Section 2.1 in order to reflect different assumptions made about failures and failure

detection. Each protocol is then described in the context of the underlying system model it assumes. We finally point out the limitations of the discussed protocols in terms of the different evaluation metrics we have used so far, namely, performance and compliance with existing commercial systems.

4.1 The Non-Blocking Atomic Commitment Problem

The *Non-Blocking Atomic Commitment* (NB-AC) problem [BHG87, Had90, BaT93] is a *fault-tolerant* agreement problem that, in addition to transaction atomicity, aims at providing transaction liveness guarantees. Formally, the NB-AC problem is defined by the *AC-Uniform-Agreement*, *AC-Uniform-Validity*, *AC-Uniform-Integrity*, *AC-Non-Triviality*, and *AC-Termination* properties of the AC problem (cf. Sections 2.2 and 2.3), and the following *AC-Non-Blocking* property:

– **AC-Non-Blocking:** Every correct participant eventually decides.

The *AC-Non-Blocking* condition is a liveness condition in the sense that it ensures *progress* at correct participants despite failures of others. Note that this property is expressed in terms of *correct* participants and not *operational* ones. This is because an operational participant might be faulty, i.e., it has crashed and then recovered, in which case, it must decide through the associated recovery protocol rather than the commit protocol [BaT93]. An atomic commitment protocol is said to be *non-blocking* (also called *fault-tolerant*) if it satisfies all of the six properties of the NB-AC problem.

Just like other fault-tolerant agreement problems, the solvability of the NB-AC problem totally depends on the nature of "admissible" faults and the ability to detect them. The latter issue is of particular importance as it is tightly dependent on the underlying system that is considered. More precisely, the ability to have a (more or less) precise knowledge about the occurrence of faults depends on the *synchrony* guarantees that the underlying system can provide. Solutions to the NB-AC problem can thus be categorized according to whether liveness guarantees are achieved assuming (1) a *synchronous* system or, on the other extreme, (2) a totally *asynchronous* system.

4.2 NB-AC in Synchronous Systems

As already stated before, solutions to fault-tolerant agreement problems in general, and to the NB-AC problem in particular, depend heavily on the assumptions made about the computational model and the kind of failures to which it is prone. In this section, we refine the general model described in Section 2.1, and consider a synchronous model of computation. We then discuss well-known non-blocking atomic commitment protocols that have been proposed in this context.

4.2.1 System Model

The model of *synchronous* computation we consider in the present and the following chapter is closely patterned after the one in [BaT93]. Synchrony is actually an attribute of both processes and communication links. A system is said to be *synchronous* if there is a known upper bound on both message transfer delays and process relative speeds.

Since it is well known that distributed systems with unreliable communication do not admit non-blocking solutions to the atomic commitment problem [Gra78, Had90, HaM90], we also assume reliable communication between the processes in the following sense: if a process P_i sends a message to a process P_k, then unless P_k is down, the message is received by P_k within δ time units after being sent, i.e., no link failures occur. The parameter δ includes the message transfer delay as well as the time required to process it at the sending and receiving processes. In such a model, site failures can be *reliably* detected and reported to any operational site by means of *timeouts*. For instance, if a process P_i does not receive an answer to a message it has sent to P_k within 2δ time units after sending the message, it can safely deduce that P_k is faulty.

4.2.2 The Three-Phase Commit Protocol

Assuming that communication is failure-free, several non-blocking atomic commitment protocols have been proposed, the most well known of which is the *Three-Phase Commit* (3PC) protocol [Ske81].

Failure-Free Execution

The 3PC protocol can be seen as a straightforward extension of 2PC. One way to understand 3PC is to understand why 2PC is blocking. In 2PC, blocking can occur because some participants may commit the transaction following the receipt of a *commit* decision while others are still uncertain about the transaction outcome[12]. Consequently, if crash failures happen in such a way that *all correct* participants are uncertain, these participants are blocked (cf. Section 2.3.2). Indeed, they cannot decide *abort* without risking a violation of the *AC-Agreement* property because some failed participants could have decided *commit*.

The idea underlying 3PC is to prevent this situation by ensuring that if any correct participant is uncertain about the transaction, then *no* participant (whether correct or not) could have decided *commit*. This is achieved by inserting an extra phase, called the *pre-commit* phase, in between the two phases of the 2PC protocol. During this phase, a preliminary decision is reached before the final decision is made.

More precisely, when the coordinator of 3PC finds that all participants' votes are *yes*, it sends a *pre-commit* message to all participants. When a participant receives *pre-commit*, it sends a *pre-commit* acknowledgment to the coordinator. By receiving the *pre-commit* message, a participant learns that all votes were *yes*, and thus, moves outside its uncertainty period. Once the coordinator has received the *pre-commit* acknowledgments from all, it decides *commit* and sends its decision to all participants. Finally, when a participant receives the *commit* decision from the coordinator, it decides *commit* (at this point, the participant knows that all other participants are outside their uncertainty period). This describes 3PC assuming no participant votes *no*, and no participant crashes during the protocol[13]. Figure 4.1 illustrates the 3PC protocol execution between six participants, P_1, P_2, P_3, P_4, P_5, and P_6, where P_1 acts as the coordinator of the protocol.

[12] Recall that an uncertain participant does not know whether or not the remaining participants have voted *yes*.

[13] Note that, to simplify the subsequent discussion and concentrate on the non-blocking feature of the described protocols, we henceforth omit from our discussion and evaluations decision acknowledgment messages.

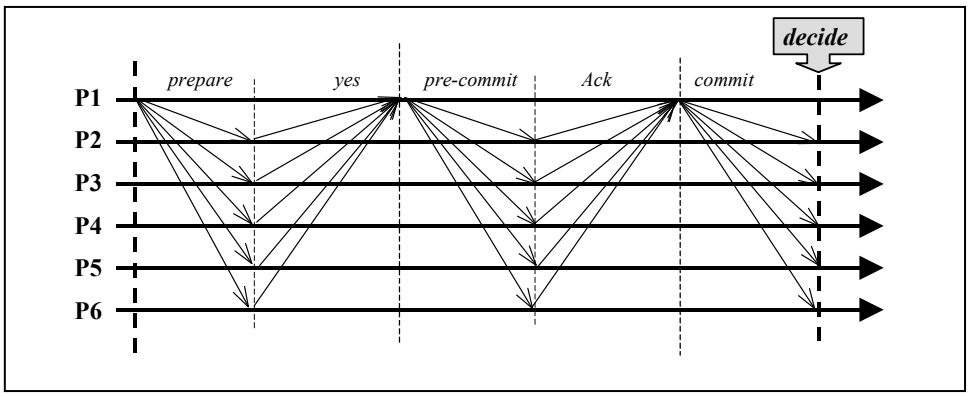

Figure 4.1: *The 3PC protocol.*

Dealing with Failures

In order to deal with failures, special timeout actions that describe what a process should do if an expected message does not arrive must be supplied. Furthermore, a recovering participant must be able to reach a decision consistent with the decision operational processes may have reached.

There are five cases to consider: (1) a participant is waiting for the *prepare* message, (2) the coordinator is waiting for *vote*s, (3) a participant is waiting for *pre-commit*, (4) the coordinator is waiting for *pre-commit* acknowledgments, and (5) a participant is waiting for *commit*. Cases (1) and (2) are handled in exactly the same way as in 2PC (cf. Section2.3.2). Case (4) means that a participant failed before sending a *pre-commit* acknowledgment. However, since the failed participant has already voted *yes*, the coordinator ignores its failure and decides *commit* as if no failure has taken place.

In cases (3) and (5), however, participants cannot decide on their own[14]. Therefore, they start a termination protocol during which they communicate with other participants to find out what to decide. In order to satisfy *AC-Non-Blocking*, the termination protocol associated with 3PC must enable all correct participants to reach a consistent decision without waiting for failures to be repaired. The basic idea of this protocol is to elect a new coordinator, called *backup coordinator*, from the set of correct participants. Once

[14] Although in case (5) participants are outside their uncertainty period, they cannot decide *commit* because some other participants might be still uncertain about the transaction outcome.

elected, the backup will direct all the correct participants toward a *commit* or an *abort* depending on its own *local state*. A participant is in an ABORT state if (i) it has already decided *abort*, or (ii) it can unilaterally decide so. It is in an UNCERTAIN state if it is in its uncertainty period. It is in a COMMITTABLE state if it has received the *pre-commit* message but not the *commit* decision. Finally, a participant is in a COMMIT state if it has already decided *commit*. The backup coordinator decides *abort* when its local state is (1) ABORT, or (2) UNCERTAIN, and decides *commit* when its local state is (3) COMMITTABLE, or (4) COMMIT:

Case (1) indicates that the backup (i) has not voted yet, or (ii) has voted *no*, or (iii) has already received an *abort* decision before the invocation of the termination protocol. In (i) and (ii), it is clear that no participant could have previously decided *commit*, while (iii) means that the 3PC coordinator had started to send *abort* decisions before it crashed. Since the coordinator sends the same decision to all participants, no participant could have received a *commit* decision, and hence, no participant could have decided *commit*. In case (2), since the *pre-commit* phase of 3PC prevents any participant from deciding *commit* once some correct participant is still uncertain, a backup with an UNCERTAIN local state is sure that no participant could have decided *commit*.

Case (3) indicates that the backup has already received a *pre-commit* message from the 3PC coordinator. This means that (i) all participants must have voted *yes*, i.e., no participant could have unilaterally decided *abort*, and (ii) no participant could have received an *abort* decision from the 3PC coordinator given that the latter had already initiated the *pre-commit* phase before it crashed, i.e., no participant could have decided *abort*. Finally, case (4) implies that the backup has received a *commit* decision from the 3PC coordinator, meaning that (i) all participants must have voted yes, i.e., no unilateral *abort*, and (ii) no participant could have received an *abort* decision from the 3PC coordinator, as the latter sends the same decision to all participants. Hence, no participant could have decided *abort*.

Since failures may occur during the termination protocol execution, a backup asks all participants to move to its local state, and waits for an acknowledgment of their state transition before sending them its final decision. This actually ensures that, in the event

of a backup crash, subsequent backups will make the same decision. From the above discussion, it follows that 3PC (and its termination protocol) satisfies the *AC-Uniform-Agreement*, *AC-Uniform-Validity*, *AC-Uniform-Integrity*, *AC-Non-Triviality*, and *AC-Non-Blocking* conditions of the NB-AC problem.

We now turn our attention to recovering participants. To satisfy *AC-Termination*, a recovering participant is required to reach a decision consistent with the decision reached by correct ones. As in 2PC, a failed participant returns to the operational state using the information it stored in its stable log. During recovery, the participant tries to decide on the transactions that were active at the time the crash occurred. This is actually achieved in exactly the same way as in 2PC. Therefore, we will not discuss the issue any further [15].

Finally, note that in an attempt to reduce the time complexity of 3PC, a decentralized 3PC variation has been also discussed in [Ske81]. Similarly to decentralized 2PC, decentralized 3PC reduces the time complexity of 3PC from 5 communication steps to 3 at the expense of a higher message complexity.

4.2.3 The ACP-UTRB Protocol

Although non-blocking, the 3PC protocol requires 5 communication steps so that a decision can be reached at every correct participant. In order to reduce the time complexity of 3PC, Babaoglu and Toueg have proposed the ACP-UTRB protocol [BaT93]. ACP-UTRB has the same basic structure as 2PC, and achieves non-blocking by exploiting the properties of the communication primitive it uses to disseminate decision messages to the participants in a transaction. The primitive that achieves this dissemination is called *broadcast*, and has a corresponding action at the destination, called *deliver*. *Broadcast* and *deliver* are usually implemented using multiple *send* and *receive* operations that the underlying network provides.

[15] Note, however, that if the participant had failed after voting *yes* but before receiving the decision, the participant needs to communicate with other processes asking them what to decide, whether or not it has already received *pre-commit*.

```
// for the broadcaster, S-broadcast(m, φ) occurs as follows:
        send(m) to all processes in φ;

// for each process in φ, S-deliver(m) occurs as follows:
        when receive(m)
            S-deliver(m);
```

Figure 4.2: *A Simple Broadcast primitive.*

Failure-Free Execution

In order to understand ACP-UTRB, let us first examine how the coordinator of a 2PC disseminates decision messages to the participants in a transaction. In 2PC, the dissemination of decision messages is achieved using a broadcast primitive, called *Simple Broadcast* (SB), which satisfies the following three properties (with $\Delta = \delta$) [BaT93]:

- **Validity:** If a correct process broadcasts a message m to the members of set φ, then all correct processes in φ eventually deliver m.
- **Uniform-Integrity:** For any message m, every correct process in φ delivers m at most once, and only if m was previously broadcast by some process.
- **Uniform-Δ-Timeliness:** There exists a known constant Δ such that if the broadcast of a message m is initiated at real-time t, then no process in φ receives m after real-time $t + \Delta$.

SB is defined in terms of two primitives, *S-broadcast(m, φ)* and *S-deliver(m)*, where m is the message broadcast to all the members of set φ. Figure 4.2 illustrates a Simple Broadcast algorithm [BaT93]. Note that SB is *unreliable*, i.e., if the broadcaster crashes while broadcasting a message m, some processes might deliver m while some *correct* processes never do so. Recall that 2PC leads to blocking situations because it allows faulty participants to decide on the transaction following the delivery of the coordinator's decision, while all correct participants never deliver that decision. Consequently, if all correct participants are uncertain, they are blocked. They cannot decide *abort* because some failed participants could have decided *commit*.

```
// for the broadcaster, R-broadcast(m, φ) occurs as follows:
        send(m) to all processes in φ;

// for each process P in φ, R-deliver(m) occurs as follows:
        when receive(m) for the first time
            if P ≠ broadcaster then send(m) to all processes in φ;
            R-deliver(m);
```

Figure 4.3: *A Uniform Timed Reliable Broadcast primitive.*

In ACP-UTRB, such blocking scenarios are prevented by using a different broadcast primitive, called *Uniform Timed Reliable Broadcast* (UTRB), which guarantees, in addition to the *Validity*, *Uniform-Integrity*, and *Uniform-Δ-Timeliness* properties of Simple Broadcast, the following *Uniform-Agreement* property:

- **Uniform-Agreement:** if any participant, correct or not, delivers a message *m*, then all correct participants in φ eventually deliver *m*.

UTRB is defined in terms of two primitives, *R-broadcast(m, φ)* and *R-deliver(m)*, where *m* is the message broadcast to all the members of set φ. Figure 4.3 describes a possible UTRB algorithm [BaT93, HaT94]. Every process relays every message it receives for the first time to all other processes, and then delivers the message. Thus, if a process delivers a message *m*, then it has already achieved relaying it. This guarantees that all correct processes will eventually deliver *m*. It is clear that this primitive satisfies *Uniform-Agreement* even if the initial broadcaster (or the *relayer*) subsequently crashes. Furthermore, in [BT93], the authors show that there exists a constant delay $\Delta = (F + 1)\delta$, by which the delivery of *m* occurs, where *F* denotes the maximum number of processes that may crash during the execution of the atomic commitment protocol.

Figure 4.4 illustrates the ACP-UTRB protocol, assuming no participant votes *no* and no participant crashes during the protocol execution. The set of participants is $\{P_1, P_2, P_3, P_4, P_5, P_6\}$, and the coordinator is P_1. The protocol is directly obtained from 2PC by replacing the SB primitive with the UTRB primitive in order to disseminate decision messages.

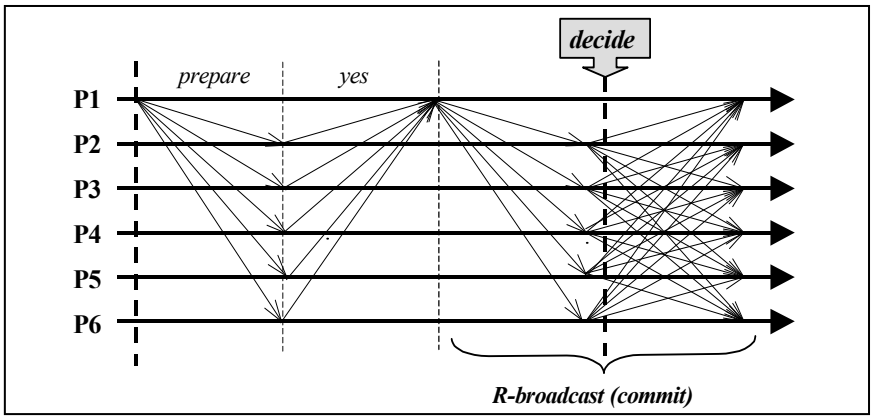

Figure 4.4: *The ACP-UTRB protocol.*

Dealing with Failures

Recall from Section 2.3.2 that the only place in 2PC where a participant cannot unilaterally decide on the transaction is when it times out waiting for the decision message. In this case, the participant starts a termination protocol during which it tries to find out what to decide by consulting with other participants in the transaction. This termination protocol may, however, lead to blocking situations if all correct participants are uncertain about the transaction outcome.

By exploiting the UTRB primitive to disseminate decision messages, ACP-UTRB eliminates the blocking scenarios of 2PC. More precisely, once a participant in ACP-UTRB has sent a *yes* vote following the receipt of *prepare*, it sets its timeout to $\delta + \Delta$, where δ represents the upper bound on the time delay needed for its vote to reach the coordinator, while Δ represents the upper bound on the time delay needed for the decision message to reach every correct participant. If, due to a coordinator crash, the participant times out while waiting for the decision message, it can unilaterally decide *abort*, safe in its knowledge that no other participant could have received *commit* (by the *Uniform- Agreement* and *Uniform-Δ-Timeliness* properties of UTRB). Thus, by substituting the termination protocol of 2PC with a unilateral *abort* decision, ACP-UTRB eliminates the only potential source of indefinite wait.

To complete our discussion on failures, note that in ACP-UTRB, participants' recovery is achieved in exactly the same way as in 2PC so that *AC-Termination* is satisfied.

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
3PC	$5n$	$2n + 3$	5	$2n + 1$
ACP-UTRB	$2n + n^2$	$2n + 1$	3	$2n + 1$

Figure 4.5: *The cost of transaction commit under 3PC and ACP-UTRB.*

4.2.3 Performance Evaluation

In this section, we examine the cost for non-blocking under the 3PC and ACP-UTRB protocols. Figure 4.5 summarizes the performances of both protocols in terms of latency and message complexity needed to commit a transaction. We denote by n the total number of participants in the transaction, and assume failure-free executions in which every participant votes *yes*.

By introducing a *pre-commit* phase, 3PC achieves non-blocking at the expense of 5 communication steps needed until a decision is reached at every correct participant, compared to 3 steps needed in blocking 2PC. Concerning message complexity, 3PC requires up to $5n$ messages (resp. $2n$+3 messages), assuming a point-to-point network (resp. a broadcast network), while $3n$ (resp. n+2) messages are exchanged under 2PC[16]. This high cost is paid even during normal processing, i.e., when no crash failures occur during the protocol execution, which is definitely unacceptable in today's highly reliable distributed platforms.

By sharing the same basic structure with 2PC, ACP-UTRB reduces the time complexity of 3PC, as it requires 3 communication steps so that a commit decision is reached at every correct participant. This comes, however, at the expense of a quadratic number of messages required by the UTRB primitive (n^2) in case of a point-to-point network, making a total of $2n+n^2$ messages exchanged during the protocol execution. However, in case of a broadcast network, ACP-UTRB outperforms 3PC in both time and message complexity.

[16] Recall that decision acknowledgment messages are not considered.

Finally, it is noteworthy that while 3PC and ACP-UTRB achieve non-blocking assuming a synchronous system and reliable communication (cf. Section 4.2.1), both protocols may result in participants reaching inconsistent decisions if either of these assumptions is not satisfied. In ACP-UTRB, for instance, unreliable communication (resp. unbounded message processing and transmission delays) renders the Uniform-Agreement property (resp. the Uniform-Δ-Timeliness property) of UTRB unattainable, leading participants to decide inconsistently in response to timeouts. Similar inconsistencies might arise under 3PC and its associated termination protocol if either of the above mentioned conditions does not hold.

To illustrate, consider a transaction involving three participants P_1, P_2 and P_3, where P_1 is the transaction coordinator. Consider the following scenario: all participants vote *yes*. P_1 receives the *yes* votes, sends *pre-commit* to all, and waits for *pre-commit* acknowledgments. Assume that, due to communication failures or arbitrary (i.e., unbounded) message transmission delays, P_3 times out waiting for the coordinator's *pre-commit* message. P_1 and P_2, on the other hand, do receive and acknowledge this message. According to the timeout actions associated with 3PC (cf. Section 4.2.2), P_3 invokes a termination protocol during which a backup coordinator, say P_3 itself, is elected. Since P_3's local state is UNCERTAIN, it decides *abort* according to the decision rule of the termination protocol. On the other hand, when P_1 times out (within the 3PC protocol) waiting for the *pre-commit* acknowledgment from P_3, it decides *commit*, given that P_3 has voted *yes*.

To avoid such inconsistencies, new 3PC variations that exploit a *quorum* (or majority) based termination protocol have been proposed [Ske82, KeD94]. More precisely, the protocols discussed in [Ske82, KeD94] guarantee that, even if the abovementioned system assumptions are not satisfied, no two participants can decide differently. Unfortunately, these protocols do not completely eliminate blocking, but they cause blocking less frequently than 2PC. However, there is no precise characterization of the conditions under which these protocols provide liveness guarantees in systems where no timing assumptions can be made whatsoever.

4.3 NB-AC in Asynchronous Systems

Based on the general system model described in Section 2.1, we define in this section an asynchronous model of computation. We then overview the most well known atomic commitment protocols that have been proposed in this context.

4.3.1 System Model

The model of *asynchronous* computation we consider in this and the following chapter is patterned after the one in [Cha93, ChT96]. Informally, a system is said to be *asynchronous* if there is no bound on message transfer delays or process relative speeds. The asynchronous model of computation is very attractive and compelling because distributed algorithms designed and implemented in this context bring general solutions to distributed problems, which are very easy to port. Furthermore, today's large distributed systems are often subject to variable or unexpected workloads that are sources of asynchrony.

Although asynchronous systems are very attractive in practice, Fischer, Lynch, and Paterson have shown that distributed agreement problems are impossible to solve in a deterministic and fault-tolerant (i.e., non-blocking) way in an asynchronous system that is subject to even a single crash failure [FLP85]. This theoretical result, known as the *Fischer-Lynch-Paterson impossibility result* (FLP, for short), applies to a variety of well-known agreement problems, notably the *Consensus* problem (cf. Section 4.3.3), and the NB-AC problem. This result translates the fact that, in an environment where no timing assumptions can be made whatsoever, it is impossible to distinguish a crashed process from a process that is only "very slow". Therefore, crash failures cannot be reliably detected and reported to correct processes.

To circumvent this impossibility result, Chandra and Toueg have augmented the asynchronous model of computation with the notion of *unreliable failure detectors* for systems with crash failures [Cha93, CT96]. More precisely, each process P_i has access to a local failure detector module FD_i, which informs it of the list of processes that it currently suspects to have crashed. A failure detector can make mistakes by providing *incorrect* information, i.e., it may suspect a correct process, or never suspect a failed one.

Furthermore, at any given time, the failure detector modules at two different processes may provide *inconsistent* information, i.e., they do not have the same list of suspects.

Although a failure detector can make mistakes, it must, however, follow a certain behavior pattern so that it can be useful. This behavior is captured through two abstract properties that the failure detector must satisfy, namely *completeness* and *accuracy*. These properties are detailed in the next section.

It is important to note that the model of unreliable failure detectors proposed in [Cha93, CT96], and which has formed the bases for the construction of existing solutions to the NB-AC problem in the context of asynchronous systems, only considered systems in which process crashes are permanent (henceforth called a *crash-stop* failure model). In Chapter 5, we show how to exploit the results presented in [Cha93, CT96] to solve the distributed commit problem in asynchronous systems in which processes may crash and later recover (henceforth called a *crash-recovery* failure model).

While processes may crash, the communication subsystem is assumed to be reliable in the following sense: if a process P_i sends a message to a process P_k, then unless one of them crashes after the message is sent, the message is eventually received by P_k [17].

4.3.2 Properties of Failure Detectors

As stated earlier, failure detectors are characterized by *completeness* and *accuracy* properties. The completeness property characterizes the degree to which a failure detector can suspect crashed processes, while the accuracy property restricts the false suspicions that a failure detector can make. Two completeness properties and four accuracy properties have been defined:

- **COMPLETENESS:**
  - ***Strong Completeness:*** Eventually, every process that crashes is permanently suspected by *every* correct process.
  - ***Weak Completeness:*** Eventually, every process that crashes is permanently suspected by *some* correct process.

[17] Note that this does not exclude link failures, assuming that these are eventually repaired so as to allow retransmission of lost or corrupted messages.

COMPLETENESS	ACCURACY			
	Strong	**Weak**	**Eventually Strong**	**Eventually Weak**
Strong	*Perfect* *P*	*Strong* *S*	*Eventually Perfect* *◇P*	*Eventually Strong* *◇S*
Weak	*Q*	*Weak* *W*	*◇Q*	*Eventually Weak* *◇W*

Figure 4.6: *Failure detector classes.*

- **ACCURACY:**
  - ***Strong Accuracy:*** No process is suspected before it crashes.
  - ***Weak Accuracy:*** Some correct process is never suspected.
  - ***Eventual Strong Accuracy:*** There is a time after which correct processes are not suspected by any correct process.
  - ***Eventual Weak Accuracy:*** There is a time after which some correct process is never suspected by any correct process.

A failure detector is characterized by the completeness property and the accuracy property that it satisfies. By combining the two completeness properties with the four accuracy properties, eight different classes of failure detectors can be defined. These are summarized in Figure 4.6. In [Cha93, ChT96], it has been shown that *Strong Completeness* can be emulated out of *Weak Completeness*, meaning that any failure detector of class *Q* (resp. *W, ◇Q,, ◇W)* can be transformed into a failure detector of class *P* (resp. *S*, *◇P*, *◇S*). Note that failure detectors satisfying *Strong Accuracy* are reliable, i.e., they never make false suspicions, whereas all other failure detectors are *unreliable,* i.e., they can make an infinite number of false suspicions.

The fundamental result of Chandra and Toueg's work on failure detectors states that the Consensus problem, an abstract form of agreement, can be solved deterministically in an asynchronous system augmented with an unreliable failure detector. The relevance of this result to our transactional context lies in the similarity between the Consensus problem and the NB-AC problem given that both problems entail fault-tolerant

agreement among processes. In [ChT96], the authors describe several solutions to the Consensus problem using each one of the eight failure detector classes. Of particular interest is an algorithm that solves Consensus using any failure detector of class $\diamond S$ and assuming a majority of correct processes, i.e., the algorithm tolerates up to f crash failures, where $f < \lceil n / 2 \rceil$. The importance of class $\diamond S$ resides in the fact that it is the *weakest* class of failure detectors that allows solving the Consensus problem in an asynchronous system [ChT96, CHT96]. With a stronger failure detector class, notably class S, the resilience of the algorithm can be increased up to n - 1.

4.3.3 A Story of Consensus

The Consensus problem can be viewed as a general form of agreement in distributed systems. In this problem, each process P_i proposes a binary initial value v_i ($v_i \in \{0, 1\}$) and the processes must agree on some binary decision value v ($v \in \{0, 1\}$) such that the following properties are satisfied [Fis83]:

- **C-Agreement:** No two correct processes decide differently.
- **C-Uniform-Validity:** The decision value must be the initial value of some process.
- **C-Uniform-Integrity:** Every process decides at most once.
- **C-Non-Blocking:** Every correct process eventually decides.

Interestingly, the algorithms proposed in [ChT96] actually solve a stronger form of Consensus, called *Uniform Consensus*. The Uniform Consensus problem is defined by the *C-Uniform-Validity*, *C-Uniform-Integrity*, and *C-Non-Blocking* properties of Consensus, and the following *C-Uniform-Agreement* property:

- **C-Uniform-Agreement:** No two processes decide differently.

Whereas the Consensus problem allows two processes to decide differently as long as at least one of them crashes, Uniform Consensus forbids any two processes from ever deciding differently whether they crash or not. This uniform agreement on the decision value is crucial for maintaining decision consistency if we consider that crashed processes may become operational again by executing a recovery protocol. This issue will be further discussed in the next chapter.

4.3.4 On the Solvability of NB-AC

The fundamental results of Chandra and Toueg on solving Consensus have constituted the cornerstone of several research works around fault-tolerant agreement problems in the context of asynchronous systems. Given these results, an interesting question is then whether the NB-AC problem can also be solved in asynchronous systems with unreliable failure detectors.

In [Gue95], Guerraoui answers this question negatively. More precisely, the author shows that NB-AC is impossible to solve in an asynchronous system with unreliable failure detectors, which is rather not surprising given that NB-AC was proved harder than Consensus [Had90]. This actually explains why NB-AC has been mostly studied under the assumption of reliable failure detection. With this impossibility, one is naturally tempted to go one step further and find out the real reason behind it.

A key result of the work presented in [Gue95] is a clear identification of the reason why NB-AC cannot be solved using unreliable failure detectors. This result states that the difficulty in solving NB-AC stems from its *AC-Non-Triviality* condition (if all participants vote *yes* and "no failures occur", then all participants must decide *commit*), which requires precise, i.e., reliable, knowledge about failures that unreliable failure detectors cannot provide. By weakening the *AC-Non-Triviality* condition, however, Guerraoui defines a weaker problem than NB-AC, called NB-WAC (*Non-Blocking Weak Atomic Commitment*), which is sufficient in most real transactional systems. A fundamental characteristic of NB-WAC is its reducibility to Consensus, i.e., whenever Consensus is solvable, NB-WAC is also solvable [Gue95].

4.3.5 The Non-Blocking Weak Atomic Commitment Problem

The *Non-Blocking Weak Atomic Commitment* (NB-WAC) problem is defined by the *AC-Uniform-Agreement*, *AC-Uniform-Validity*, *AC-Uniform-Integrity*, and *AC-Non-Blocking* of the NB-AC problem, and by the following *AC-Weak-Non-Triviality* condition [Gue95]:

- **AC-Weak-Non-Triviality:** If all participants vote *yes* and no participant is ever suspected, then all participants must decide *commit*.

The importance of this new condition lies in the fact that, although weaker than its original version (i.e., transactions are allowed to abort in case of failure *suspicions*), it still eliminates trivial solutions to the problem where participants always decide *abort*. As stated before, a fundamental characteristic of NB-WAC is that it is reducible to Consensus, and therefore, is solvable in asynchronous systems augmented with unreliable failure detectors. The main significance of this result is in defining a rigorous framework in which atomic commitment with some liveness guarantees can be achieved.

Note, however, that although solvable in the context of asynchronous systems, NB-WAC was proposed assuming a *crash-stop* failure model, i.e., once a process crashes, it does not recover. This assumption is translated by the absence of the *AC-Termination* property from the set of properties that define the NB-WAC problem.

4.3.6 The DNB-AC protocol

Based on the above results, several protocols that solve the NB-WAC problem were devised and are typified by the *Decentralized Non-Blocking Atomic Commitment* (DNB-AC) protocol [GuS95]. In the absence of failure suspicions, DNB-AC has the same basic structure as the decentralized 3PC protocol discussed by Skeen in the context of synchronous systems [Ske81] (cf. Section 4.2.2). As opposed to decentralized 3PC, however, the termination protocol of DNB-AC is encapsulated within a *uniform consensus* protocol, enabling a precise characterization of its liveness in an asynchronous system augmented with any unreliable failure detector of class $\diamond S$.

As illustrated in Figure 4.7, the DNB-AC protocol has three communication steps. During the first step, the coordinator initiates the protocol by sending a *prepare* message to all participants in the transaction. In step 2 of the protocol, a participant that votes *yes* sends its vote to all other participants. In step 3, when a participant receives *yes* votes from all, it sends a *pre-commit* message to all. Finally, once a participant has received *pre-commit* from all, it decides *commit*. Note, however, that a participant that decides on the transaction needs to forward its decision to all other participants. This is required in order to ensure that if a correct participant reaches a decision, then all correct participants also reach a decision.

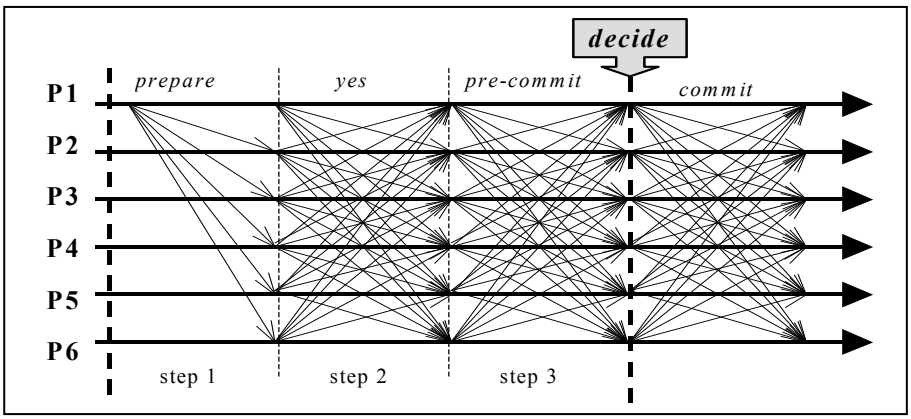

Figure 4.7: *The DNB-AC protocol.*

This describes the protocol assuming no participant votes *no*, and no participant is suspected to have crashed during the protocol execution. If, during the first step of the protocol, a participant P_i either suspects the coordinator or votes *no*, then P_i takes a unilateral *abort* decision, and sends *abort* to all other participants. During step 2, however, a participant P_i that suspects any other participant cannot unilaterally decide. Therefore, P_i asks all other participants to start a uniform consensus protocol by sending them a *start-consensus* message, and then starts the uniform consensus with *abort* as its initial value. This value translates the fact that at this point, P_i does not know yet the votes of all participants. The outcome of the uniform consensus protocol defines the transaction outcome for P_i. During step 3 of the protocol, if a participant P_i suspects any other participant or receives a *start-consensus* message, then P_i starts a uniform consensus with *commit* as its initial value (at this point, P_i knows that all votes are *yes*), and the outcome of the consensus protocol becomes the transaction outcome for P_i.

In the absence of failure suspicions, it is clear that DNB-AC preserves transaction atomicity as it reduces to a classical decentralized 3PC protocol. In the event of failure suspicions, DNB-AC exploits a uniform consensus protocol as a termination protocol, which guarantees a unique outcome for the transaction in a fault-tolerant way. It follows that the resilience (to blocking) of DNB-AC depends on the resilience of the uniform consensus protocol, and hence on the underlying failure detector class that is considered. More precisely, based on a failure detector of class $\diamond S$, DNB-AC tolerates up to f crash failures, where $f < \lceil n / 2 \rceil$, i.e., at least $\lceil (n + 1) / 2 \rceil$ participants are correct.

4.3.7 The Modular Decentralized 3PC Protocol

While DNB-AC needs the same number of communication steps to commit as blocking 2PC protocols (i.e., 3 steps), $3n^2+n$ (resp. $3n+1$) messages need to be exchanged during the protocol execution assuming a point-to-point (resp. a broadcast) network. It is in an attempt to reduce the message complexity associated with DNB-AC that the *Modular Decentralized 3PC* (MD3PC) protocol has been proposed [GLS96].

The key idea underlying MD3PC is to have the sub-protocol required for non-blocking performed by only a subset noted Set_{NB} of the participants in the transaction, and the cardinality of this subset depends on the number of crash failures to be tolerated. As a consequence, the resilience of the protocol is traded against the number of messages exchanged during its execution. More precisely, to be resilient to f crash failures, given that $f < \lceil n / 2 \rceil$ and the failure detector is $\diamond S$, the protocol requires that Set_{NB} contain $2f+1$ members (i.e., $| Set_{NB} | = 2f + 1$). Another fundamental difference with DNB-AC relates to the termination protocol used in case of failure suspicions. Whereas DNB-AC requires a uniform consensus protocol as a termination protocol, MD3PC is based on a *majority consensus*.

The Majority Consensus problem is defined by the *C-Uniform-Agreement, C-Uniform-Integrity*, and *C-Non-Blocking* properties of Uniform Consensus, and the following *C-Majority-Uniform-Validity* property:

- **C-Majority-Uniform-Validity:** (i) the decision value must be the initial value of some process, and (ii) if a majority of initial values are 1, then the decision value must be 1.

In our transactional context, the value 1 clearly corresponds to *commit* while 0 corresponds to *abort*. As opposed to the *C-Uniform-Validity* property of Uniform Consensus, the *C-Majority-Uniform-Validity* property enables a participant in MD3PC to decide *commit* once it has received *pre-commit* messages from a majority of Set_{NB}. Note that, just like Consensus and Uniform Consensus, the Majority Consensus problem can also be solved with any failure detector of class $\diamond S$ [GLS96].

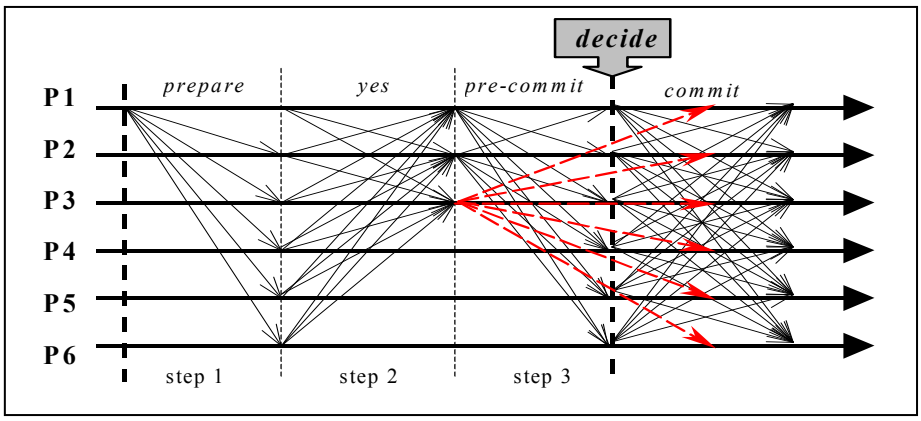

Figure 4.8: *The MD3PC protocol.*

Figure 4.8 illustrates the MD3PC protocol, assuming no participant votes *no* and no participant is ever suspected. Similarly to DNB-AC, during the first step of MD3PC, the coordinator sends a *prepare* message to all participants in the transaction. In step 2, however, participants' votes are only sent to the members of Set_{NB}. In Figure 4.8 for instance, $f = 1$ and $Set_{NB} = \{P_1, P_2, P_3\}$ (i.e., $| Set_{NB} | = 3$). In step 3, when a member of Set_{NB} receives *yes* votes from all, it sends a *pre-commit* message to all. Finally, once a participant has received *pre-commit* from a majority of Set_{NB}, it sends its decision to all other participants and decides *commit*.

If, during step 1, a participant P_i either votes *no* or suspects the coordinator, then P_i takes a unilateral *abort* decision. The remaining failure suspicion cases can be reduced to a majority consensus protocol, which is only launched by the members of Set_{NB}. More precisely, a failure suspicion that occurs during step 2 of the protocol leads a member P_i of Set_{NB} to start a majority consensus with *abort* as its initial value, while a failure suspicion that occurs during step 3 leads P_i to start a majority consensus with *commit* as initial value.

4.3.8 Performance Evaluation

In this section, we examine the cost for non-blocking under DNB-AC and MD3PC. Figure 4.9 summarizes the performances of both protocols in terms of latency and message complexity, assuming no participant votes *no* and no participant is ever suspected during the protocol execution. We denote by n the total number of participants, and by f the number of crash failures to be tolerated, where $f < \lceil n / 2 \rceil$.

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
DNB-AC	$n + 3n^2$	$3n + 1$	3	--
MD3PC	$n\,(4\,f + 3) + n^2$	$2\,(n + f + 1)$	3	--

Figure 4.9: *The cost of transaction commit under DNB-AC and MD3PC.*

As already pointed out, DNB-AC and MD3PC have the same basic structure and differ only in the number of messages that need to be exchanged during the protocol. More precisely, both protocols need, like 2PC, 3 communication steps until a decision is reached at every correct participant. In MD3PC, however, the resilience of the protocol can be traded against the number of messages exchanged. For instance, with $n = 12$, $f = 2$, and assuming a point-to-point network, DNB-AC requires 444 messages, while MD3PC requires 276 messages (compared to 36 in 2PC). In case of a broadcast network, DNB-AC needs 37 messages whereas MD3PC needs 30 (compared to 14 in 2PC).

Even though both protocols have the same latency, a participant in MD3PC is allowed to decide *commit* once it has received *pre-commit* from a majority of Set_{NB}, whereas a participant in DNB-AC cannot decide *commit* until it receives *pre-commit* from *all* participants in the transaction.

Based on Chandra and Toueg's work on solving (Uniform) Consensus, DNB-AC and MD3PC achieve non-blocking in asynchronous systems assuming reliable communication, any unreliable failure detector of class $\diamond S$, and a maximum of f crash failures, where $f < \lceil n / 2 \rceil$. If these assumptions are not satisfied, both protocols might block, but never lead two participants to decide on different outcomes.

An important point to note is that both DNB-AC and MD3PC were devised assuming a *crash-stop* failure model, meaning that a process that crashes is not assumed to recover nor to inquire other participants about the transaction outcome. Consequently, no log force is performed and no acknowledgment of decision messages is needed, as these are usually required in order to support recovery.

4.4 Discussion

In an attempt to provide transaction liveness guarantees, fault-tolerant (i.e., non-blocking) commit protocols have emerged. As shown in Sections 4.2.4 and 4.3.8, however, fault-tolerance has a price, and this price is paid in terms of time complexity, message complexity, or both. Indeed, when compared to their blocking counterparts, the commit protocols discussed in this chapter trade performance for fault-tolerance. This is mainly due to the fact that most of the existing works on fault-tolerant commit protocols have, in a way, pushed performance issues into the background, and if not, the best they hoped for is to attain performances comparable to those of blocking 2PC variations.

Another major problem has to do with participants' *prepared* states. Indeed, all the protocols discussed in this chapter are extensions of basic 2PC. Therefore, they all require that the participating local sites provide a *prepared* state for each transaction they execute, thus inheriting all the problems associated with the support of that state (cf. Sections 1.2 and 2.6).

Finally, and as already pointed out, solutions to the NB-AC problem depend on the underlying system and failure assumptions, and on the knowledge about the occurrence of failures in the system. Whereas this knowledge can be precise under a synchronous system, asynchronous systems render any such knowledge imprecise, and thus any solution to the NB-AC problem impossible. By considering a slightly weaker variation of NB-AC, fault-tolerant commit protocols have started to emerge in the context of asynchronous systems (cf. Sections 3.4.6 and 3.4.7). These protocols essentially build on the work of Chandra and Toueg on solving Consensus in asynchronous systems, and hence assume, just like consensus protocols, a failure model in which process crashes are permanent. Whereas this assumption indeed makes sense in environments where process decisions are used to trigger some real-time actions, i.e., there is no time to take into account process recovery and hence the decision of faulty processes, it is definitely unacceptable in a transactional context where participants' recovery is an intrinsic feature, and where the decision of faulty processes must be taken into account if transaction atomicity is to be guaranteed.

Chapter 5

Non-Blocking Dictatorial Atomic Commitment

As we have shown in the previous chapter, existing non-blocking commit protocols impose high costs on distributed transaction processing, which results in a significant increase in transaction response times. Furthermore, given the assumptions they make about the underlying system model, notably in the context of asynchronous systems, their practical utility in real-world transactional systems becomes questionable. Based on the observation that all these protocols are extensions of 2PC, an important question is then whether non-blocking protocols can rather be derived from 1PC, hence reconciling high performance and fault-tolerance. In this chapter, we answer this question positively, and propose several non-blocking solutions to the Dictatorial Atomic Commitment problem.

In order to do so, we first discuss the issue of blocking in 1PC, and define the *Non-Blocking Dictatorial Atomic Commitment* (NB-DAC) problem. We then give a protocol, called NB-CLL, that solves the problem in a synchronous system, while maintaining the cost of distributed transaction commit below that of all existing non-blocking protocols proposed in this context.

We point out the fact that, just like the NB-AC problem, NB-DAC is unattainable in asynchronous systems. By refining the NB-DAC specification, and following the approach proposed in [Gue95], we introduce the *Non-Blocking Weak Dictatorial Atomic Commitment* (NB-WDAC) problem that is better suited to asynchronous environments. We then propose a protocol, called ANB-CLL, which solves NB-WDAC in an

asynchronous system augmented with an unreliable failure detector. In contrast with existing non-blocking protocols previously proposed in this context, our protocol achieves non-blocking in systems in which processes may crash and later recover (i.e., *crash-recovery* failure model), making it more suitable for real-world transactional systems where process recovery is an intrinsic feature.

The NB-CLL and ANB-CLL protocols can be viewed as non-blocking extensions of CLL, our 1PC variation (cf. Section 3.5). Consequently, they both blend the advantages of CLL with fault-tolerance. We show through performance analysis that our protocols are more efficient than all other non-blocking protocols proposed in their respective contexts.

5.1 The Window of Vulnerability to Blocking of 1PC

To better illustrate the blocking problem in 1PC, let us go back over this issue in 2PC. In 2PC, blocking can occur if the coordinator crashes after the participants have sent a *yes* vote. This period of time is called the *window of vulnerability* to blocking of the protocol.

In a 1PC protocol, the window of vulnerability is much larger than in a 2PC protocol. This is because the only period during which a participant has the freedom to unilaterally abort a transaction is after receiving an operation from the coordinator and before acknowledging this operation. Otherwise, the participant is at the coordinator's mercy, and the latter acts as a dictator for choosing the transaction outcome. In other words, whenever it has acknowledged an operation and unless it receives another operation or the final decision, a participant in 1PC enters the window of vulnerability to blocking. If the coordinator crashes while all participants have acknowledged the operations submitted to them, they are all blocked until the coordinator recovers from its crash. In comparison, unless it has sent back a *yes* vote, a participant in a 2PC can at any time unilaterally abort a transaction.

The above observation raises a crucial issue and suggests that non-blocking solutions to the DAC problem are indispensable if dictatorial transaction processing is to be used in today’s systems and applications.

5.2 The Non-Blocking Dictatorial Atomic Commitment Problem

We define the *Non-Blocking Dictatorial Atomic Commitment* (NB-DAC) problem by the *DAC-Uniform-Agreement*, *DAC-Uniform-Integrity*, and *DAC-Termination* properties of the DAC problem (cf. Sections 3.1.2 and 3.5.2), and the following *DAC-Uniform-Validity*, and *DAC-Non-Blocking* properties:

- **DAC-Uniform-Validity:** If the coordinator does not crash, the decision value is the coordinator's proposed value.
- **DAC-Non-Blocking:** Every correct participant eventually decides.

Just like *AC-Non-Blocking*, the *DAC-Non-Blocking* property is expressed in terms of correct participants and not operational ones. This is mainly due to the fact that, in a transactional context, operational participants that have crashed and later recovered should decide through the associated recovery protocol rather than the commit protocol. As we shall see later, it is precisely this feature that enables us to extend the applicability field of Chandra & Toueg's unreliable failure detectors model so as to solve the distributed commit problem in asynchronous systems based on a *crash-recovery* model of computation.

5.3 NB-DAC in Synchronous Systems

Based on the system model described in Section 4.2.1, we propose in this section a protocol, called *Non-Blocking Coordinator Logical Log* (NB-CLL), which solves the NB-DAC problem in synchronous systems [AbP98a, AbP98b]. We then prove the correctness of our protocol and compare its performances with existing non-blocking protocols proposed in this context.

5.3.1 The NB-CLL Protocol

Failure-Free Execution

NB-CLL has exactly the same basic structure as the CLL protocol (cf. Section 3.5), and differs only in the way decision messages are disseminated by the coordinator of the transaction. To illustrate, recall that CLL (as well as all other 1PC variations) exploits a basic 1PC protocol, which we defined in terms of the *terminate()* function in Figure 3.1.

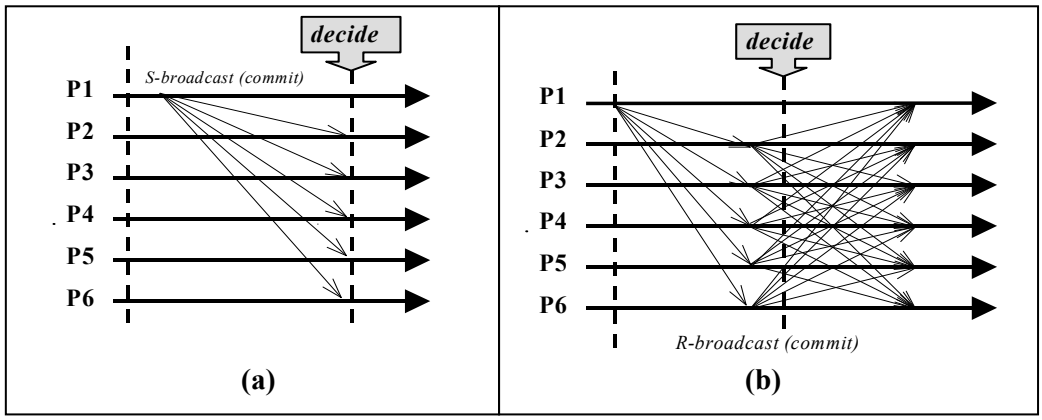

Figure 5.1: *(a) The CLL protocol, and (b) the NB-CLL protocol.*

During this function, the coordinator disseminates its decision message by sequentially sending this message to each participant in the transaction. Once a participant receives the coordinator's decision, it immediately decides accordingly and returns. This message diffusion corresponds exactly to the Simple Broadcast (SB) primitive discussed in Section 4.2.3. For the sake of clarity, Figure 5.1(a) illustrates the CLL protocol behavior based on SB, and assuming no failures occur during the protocol execution. The set of participants is $\{P_1, P_2, P_3, P_4, P_5, P_6\}$, and the coordinator is P_1.

Thus, just like 2PC, 1PC protocols (including CLL) lead to blocking situations because of the unreliability of SB that allows faulty participants to deliver the coordinator's decision (and then crash), while all correct participants never deliver that decision. If failures occur such that all correct participants are uncertain, they cannot decide on the transaction even if they know that participants they cannot communicate with have crashed. Indeed, any such decision might contradict the decision another participant might have reached before crashing. Unlike 2PC, however, such blocking scenarios are much more likely to occur in a 1PC protocol given that the uncertainty period of a 1PC participant lasts all along the transaction execution.

CLL can thus be made non-blocking by substituting the SB primitive by a Uniform Timed Reliable Broadcast (UTRB) that achieves uniform agreement on decision delivery among participants [BaT93, HaT94] (cf. Section 4.2.3). Using UTRB guarantees that if any participant, whether correct or not, delivers a decision message, then all correct

participants will deliver that message within $\Delta=(F + 1)\delta$ time units after the time the coordinator has initiated the broadcast[18]. Figure 5.1(b) illustrates the resulting protocol, which we call *Non-Blocking CLL* (NB-CLL) [AbP98a, AbP98b], assuming no failures occur during the protocol execution.

Dealing with Failures

Recall that in CLL, if a participant P_i times out while waiting for a transaction's operation or the final decision from the coordinator, it cannot unilaterally decide on the transaction. In this case, P_i starts a termination protocol during which it tries to consult with other participants that might have reached a decision or can unilaterally do so. If, however, all the participants with which P_i can communicate are uncertain, P_i is blocked inside the termination protocol.

By exploiting the properties of the UTRB primitive, NB-CLL eliminates such undesirable scenarios. The idea is inspired from the ACP-UTRB protocol, and consists on substituting the (blocking) termination protocol executed in response to the timeout with an action that *always* enables a consistent decision to be reached at P_i. For this to work, however, P_i needs a *reliable* and *accurate* detection of the crash of the coordinator before it engages in the associated timeout action (otherwise, transaction atomicity would be compromised). Whereas this issue is rather straightforward in 2PC – a participant that times out waiting for the coordinator's decision in response to its vote can safely conclude that the coordinator has crashed[19] – it is less obvious in our 1PC context given that a 1PC participant does not wait for the decision (or a transaction's operation) as a response to a message it has sent to the coordinator. Therefore, there is no mean by which the participant can tell the moment at which the coordinator is supposed to terminate the transaction and broadcast its decision. As a consequence, the timeout of a 1PC participant cannot be relied on to detect a coordinator crash: the coordinator can be simply busy executing operations on other participants.

[18] Recall that F denotes the maximum number of participants that may crash during the execution of the commit protocol, while δ represents the upper bound on message processing and transport delay over a link.

[19] Recall that in our model of synchronous system and reliable communication, if a 2PC participant does not deliver the decision message within $\delta+\Delta$ time units after sending its vote (where the value of Δ depends on the particular broadcast primitive that is used), it can safely conclude that the coordinator is faulty.

To overcome this problem, we propose to augment our synchronous system model with an external failure detector mechanism by which crash failures are *reliably* detected and reported to operational sites. In other words, each process P_i has access to a reliable failure detector module RFD_i, which maintains a list of those processes that have crashed. Given our model of synchronous system and reliable communication, reliable failure detectors can be easily implemented by means of timeouts. For instance, each failure detector module RFD_i can periodically query other processes in the system. If a process P_j does not respond by the specified timeout, RFD_i can safely conclude that P_j has crashed. In the notations, $P_j \in RFD_i$ means that process P_i has detected the crash of process P_j.

In this context, our NB-CLL protocol, defined by *the terminate()* function in Figure 5.2, works as follows [AbP98a, AbP98b]. When a participant P_i detects a coordinator crash, it sets its timeout to $\Delta=(F + 1)\delta$, which represents the upper bound on the time delay needed for the decision message to reach every correct participant under UTRB. On timeout, P_i takes a unilateral *abort* decision, safe in its knowledge that no other participant could have received (and decided) *commit*.

To complete our discussion on failures, note that NB-CLL exploits our Coordinator Logical Logging recovery mechanism described in Section 3.4.3. Therefore, participant's recovery is achieved in exactly the same way as in CLL, that is, using CLL's recovery procedure of Figure 3.2 (cf. Section 3.5.2).

5.3.2 Protocol Correctness

In this section, we show that our NB-CLL protocol presented in Figure 5.2 is correct and non-blocking. This amounts to proving that it satisfies all of the five properties of the NB-DAC problem.

Theorem 5.1. *NB-CLL achieves the DAC-Uniform-Agreement property.*

PROOF. For contradiction, assume that a participant P_i decides *commit*, while another participant P_k decides *abort*. In NB-CLL, P_i can decide *commit* only at lines 7 and 13 following the delivery of a *commit* decision message. By the Uniform-Integrity property of UTRB, the coordinator must have broadcast a *commit* decision

```
function terminate ()

Only the coordinator executes:
 1  decision: = proposition;                    // proposition ∈ {commit, abort}
 2  R-broadcast (decision, participants);


Every participant P_i executes:
 3  wait until [R-deliver (decision) or coordinator ∈ RFD_i]
 4       if (coordinator ∈ RFD_i) then
 5            set time-out to Δ;
 6            wait until [R-deliver (decision)]
 7                 decide (decision);
 8                 return;
 9            on-timeout
10                 decide (abort);
11                 return;
12       else
13            decide (decision);
14            return;
```

Figure 5.2: *The NB-CLL protocol.*

message at line 2, say at real-time $t_{R\text{-}broadcast}$. Participant P_k can decide *abort* at lines 7, 10, and 13. Since we have a single coordinator per transaction and since the coordinator broadcasts only one decision for each transaction (at line 2), participant P_k could not have delivered an *abort* decision, and hence, could not have decided *abort* at lines 7 or 13. Therefore, P_k must have decided *abort* at line 10 following the time-out expiration. In this case, P_k must have detected a coordinator crash. Assuming that this detection occurs at real-time t_{crash}, the time-out expiration occurs at real-time $t_{crash}+\Delta$. Since participant P_i has delivered a *commit* decision, this means that the coordinator had broadcast a *commit* decision before it crashed. Thus, $t_{R\text{-}broadcast} < t_{crash}$. Furthermore, by the Uniform-Agreement and Δ-timeliness properties of UTRB, P_k eventually delivers a *commit* decision as well, and it does so at most by real-time $t_{R\text{-}broadcast}+\Delta$. Since $t_{R\text{-}broadcast} < t_{crash}$, P_k must have received the *commit* decision before timing out. This contradicts the fact that P_k has executed line 10.

Theorem 5.2. *NB-CLL achieves the DAC-Uniform-Validity property.*

PROOF. Assume that the coordinator is correct. Since we consider a reliable failure detector, then no participant could have detected a coordinator crash, and therefore, no participant could have decided at line 7 or at line 10. Consequently, all participants must have decided at line 13 following the delivery of the decision message, and this message must have been broadcast by the coordinator at line 2 (by the Uniform-Integrity property of UTRB). From lines 1 and 2 of the protocol, it is obvious that the decision value broadcast by the coordinator is nothing but its proposition. Consequently, the decision value of all participants is the coordinator's proposed value.

Theorem 5.3. *NB-CLL achieves the DAC-Uniform-Integrity property.*

PROOF. From the structure of the protocol, it is obvious that every participant decides at most once.

Theorem 5.4. *NB-CLL achieves the DAC-Non-Blocking property.*

PROOF. In NB-CLL, if the coordinator does not crash, then it eventually broadcasts its decision to all participants by executing line 2. Consequently, every correct participant eventually decides at line 13 following the delivery of the coordinator's decision message. On the other hand, if the coordinator crashes, then every undecided (correct) participant will eventually detect the coordinator crash, in which case, the participant executes the associated wait statement at line 6 after having set its timeout. If the decision message being waited for is not received by the specified time, the timeout expires and the participant decides *abort* at line 10; otherwise, the participant decides on line 7 following the delivery of the decision message. Therefore, every correct participant eventually decides.

Theorem 5.5. *NB-CLL achieves the DAC-Termination property.*

PROOF. To show that *DAC-Termination* is satisfied, we must consider participants' recovery. Given that NB-CLL exploits the same recovery procedure as CLL, the proof remains the same for both protocols (cf. Section 3.5.3).

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
3PC	$5n$	$2n + 3$	5	$2n + 1$
ACP-UTRB	$2n + n^2$	$2n + 1$	3	$2n + 1$
NB-CLL	n^2	n	1	$n + 1$

Figure 5.3: *The cost of transaction commit under 3PC, ACP-UTRB, and NB-CLL.*

5.3.3 Performance Evaluation

In this section, we examine the cost for non-blocking under the NB-CLL protocol, and compare its performances with previously discussed non-blocking protocols proposed in the same context, namely 3PC and ACP-UTRB. Figure 5.3 summarizes the performances of the protocols in terms of latency and message complexity needed to commit a transaction. We denote by n the total number of participants in the transaction.

By sharing the same basic structure with CLL, NB-CLL drastically reduces the time and log complexities of both 3PC and ACP-UTRB, thereby reducing transaction response times. Furthermore, we note that although NB-CLL achieves non-blocking at the expense of a quadratic number of messages exchanged under a point-to-point network, it still maintains message complexity far below that of ACP-UTRB, and with a reasonable number of participants (i.e., $n < 5$) or in the case of a broadcast network, even below that of 3PC, thus providing the best tradeoff between performance and fault-tolerance.

5.4 NB-DAC in Asynchronous Systems

Our NB-CLL protocol described in the previous section provides both safety and liveness guarantees, assuming a synchronous system and reliable communication. Just like 3PC and ACP-UTRB, however, NB-CLL may lead participants to reach inconsistent decisions if either of these assumptions is not satisfied, thus compromising transaction safety. To avoid such inconsistencies, and based on the asynchronous system model defined in Section 4.3.1, we study in this section the NB-DAC problem in asynchronous environments. In particular, we propose a new non-blocking extension to CLL, called

Asynchronous Non-Blocking Coordinator Logical Log (ANB-CLL), which *always* guarantees transaction safety, while providing liveness guarantees in asynchronous systems with reliable communication and unreliable failure detectors [AbP99].

In contrast with existing fault-tolerant protocols proposed in this context, ANB-CLL achieves non-blocking in asynchronous environments in which processes may crash and later recover. Indeed, by exploiting the recovery semantics of the distributed commit problem, we show that the previous results of Chandra & Toueg on solving (Uniform) Consensus in asynchronous systems assuming a *crash-stop* failure model [Cha93, ChT96] can be adapted to a transactional context so as to provide fault-tolerant solutions to the distributed commit problem in asynchronous systems based on a *crash-recovery* failure model of computation.

5.4.1 On the Solvability of NB-DAC

Recall from Section 5.2 that the NB-DAC problem is defined by the *DAC-Uniform-Agreement*, *DAC-Uniform-Integrity*, *DAC-Termination* and *DAC-Non-Blocking* properties, and the following *DAC-Uniform-Validity* property:

- **DAC-Uniform-Validity:** If the coordinator does not *crash*, then the decision value is the coordinator's proposed value.

This property is of particular importance as it reflects the dictatorial aspect of the NB-DAC problem as opposed to the classical NB-AC problem — unless the coordinator *crashes*, the decision value must *only* be determined by the coordinator. Clearly, this property is too strong in the context of asynchronous systems since it requires a precise knowledge about the occurrence of a coordinator crash, thus making the NB-DAC problem rather unattainable.

To illustrate, assume that the coordinator crashes before sending its decision to the participants. In this case, participants can neither wait indefinitely for the coordinator's decision (otherwise, *DAC-Non-Blocking* would be violated), nor can they take a unilateral decision unless they know that the coordinator is indeed faulty (otherwise, *DAC-Uniform-Validity* would be compromised). Since unreliable failure detectors can

never provide participants with such a precise knowledge about the crash of the coordinator, it follows that NB-DAC cannot be solved in asynchronous systems with unreliable failure detectors.

However, by weakening the *DAC-Uniform-Validity* condition, and following the approach proposed in [Gue95], we define in the next section the *Non-Blocking Weak Dictatorial Atomic Commitment* (NB-WDAC) problem that is better suited to asynchronous environments.

5.4.2 The Non-Blocking Weak Dictatorial Atomic Commitment Problem

We define the *Non-Blocking Weak Dictatorial Atomic Commitment* (NB-WDAC) problem by the *DAC-Uniform-Agreement*, *DAC-Uniform-Integrity*, *DAC-Termination*, and *DAC-Non-Blocking* properties of the DAC problem, and the following *DAC-Weak-Uniform-Validity* property:

- **DAC-Weak-Uniform-Validity:** If the coordinator is not *suspected*, then the decision value is the coordinator's proposed value.

Note that, although weaker than its original version (i.e., participants are allowed to decide unilaterally on the transaction if the coordinator is *suspected* to have crashed), this new property still maintains the dictatorial aspect of NB-DAC, while making the NB-WDAC problem solvable in asynchronous systems with unreliable failure detectors.

5.4.3 NB-(WD)AC in the Crash-Recovery Model

As we have seen in the previous chapter, the results of Chandra & Toueg on solving (Uniform) Consensus with unreliable failure detectors [Cha93, CT96] have constituted the bases for the construction of fault-tolerant solutions to the distributed commit problem in asynchronous environments. Indeed, by encapsulating failure suspicions scenarios within a uniform consensus protocol, non-blocking commit protocols have started to emerge in this context [GuS95, GLS96] (cf. Sections 4.3.6 and 4.3.7).

However, given that the consensus protocols proposed in [Cha93, CT96] have been devised assuming a *crash-stop* failure model, existing non-blocking commit protocols that build on these works follow the same assumption, and hence do not support process recovery. Whereas this assumption indeed makes sense in environments where process decisions are used to trigger some actions within a critical real-time deadline[20] (i.e., there is no time to wait for crashed processes to recover and decide, so faulty processes are simply ignored) [Had90], it is definitely unacceptable in a transactional context where process recovery is an inherent feature, and where the decision of faulty processes must be taken into account if transaction atomicity is to be preserved. This requirement is even intrinsic to the specification of the distributed commit problem and is expressed in terms of the *(D)AC-Termination* property (cf. Sections 2.3.2 and 3.5.2), which states that once a crash failure is repaired, the recovering participant must attempt to reach a consistent decision — if not immediately, then once enough failures are repaired.

In light of the above discussion, a fundamental question is then whether Chandra & Toueg's results on solving Consensus in asynchronous systems assuming a *crash-stop* failure model [Cha93, CT96] can be exploited to devise non-blocking solutions to the distributed commit problem (in its various forms) in asynchronous systems in which processes may crash and later recover.

In contrast with initial intuition, and based on the *(D)AC-Non-Blocking* property, we show that the answer to this question is "Yes" [AbP99]. To illustrate, recall that *(D)AC-Non-Blocking* requires that "every *correct* participant eventually decides". The fact that this property is expressed in terms of *correct* participants and not *operational* ones means that an operational participant that has crashed and later recovered is not allowed to participate again in the execution of the commit protocol. Instead, recovering participants have to decide through the associated recovery protocol rather than the commit protocol.

Seen in this light, the following idea for exploiting the results in [Cha93, CT96] in our transactional context while taking into account participants' recovery suggests itself. When failure suspicions occur during the execution of the commit protocol, a uniform

[20] Examples are process control systems for power plants, air traffic control, etc.

consensus protocol is launched in order to terminate the transaction in a non-blocking way at all correct participants. If, however, a participant P_i crashes while executing the uniform consensus protocol, it will not be allowed to participate again in the protocol execution in case it recovers from its crash. Instead, P_i will try to decide on the transaction inside its recovery procedure. Therefore, upon recovering, P_i informs all other participants in the transaction that, although operational, it is faulty and hence has to be excluded from any consensus protocol execution. As far as uniform consensus is concerned, this approach reduces the problem to the case where process crashes are permanent.

We conclude that the uniform consensus protocols described in [Cha93, CT96] can be adapted to our transactional context so as to provide non-blocking solutions to the distributed commit problem, while taking into account participants' recovery.

5.4.4 The ANB-CLL protocol

Based on the above results, we propose in this section the *Asynchronous Non-Blocking Coordinator Logical Log* (ANB-CLL) protocol, which solves the NB-WDAC problem in asynchronous systems assuming any unreliable failure detector of class $\diamond S$ and a majority of correct participants [AbP99].

ANB-CLL: Overview

Just like NB-CLL, ANB-CLL can be viewed as a non-blocking extension to CLL. Unlike NB-CLL, however, ANB-CLL necessitates an additional communication step so that a *commit* decision can be reached at every correct participant. As illustrated in Figure 5.4, ANB-CLL operates in two communication steps. During the first step of the protocol, if the coordinator's proposition is *commit*[21], the coordinator sends a *start-pre-commit* message to all participants. In step 2, when a participant receives *start-pre-commit* from the coordinator, it sends a *pre-commit* message to all. Finally, when a participant receives *pre-commit* from all, it decides *commit*. Note that a participant that decides on the transaction

[21] Recall that the coordinator's proposition is *commit* if (1) it has received acknowledgment messages for all the transaction's operations, and (2) it has succeeded in saving these operations on stable log.

needs to forward its decision to all other participants. This actually ensures that if a correct participant reaches a decision, then all correct participants also reach a decision.

This describes the protocol assuming the coordinator proposes *commit*, and no participant is suspected to have crashed during the protocol execution. If, during step 1, the coordinator proposes *abort*, then it sends an *abort* decision to all participants in the transaction and decides *abort*. All failure suspicion scenarios are handled within a uniform consensus protocol used as a termination protocol, enabling a consistent decision to be reached at every correct participant in a non-blocking way. More precisely, if during step 1, a participant P_i suspects the coordinator, P_i starts a uniform consensus protocol with *abort* as its initial value (at this point, the participant does not know whether the transaction has been successfully executed, i.e. all the transaction's operations have been acknowledged and the coordinator has force-written its log on stable storage). In step 2, if a participant P_i suspects any other participant, it starts a uniform consensus protocol with *commit* as its initial value (at this point, P_i knows that the transaction has been successfully executed).

ANB-CLL: Detailed Description

The ANB-CLL protocol is defined by the *terminate()* function described in Figure 5.5. This function consists of two concurrent tasks, *Task 1* and *Task 2*, and terminates by the execution of a return (*decision*) statement, by which the participant decides the value "*decision*" (and stops). To deal with failure suspicions, a uniform consensus protocol defined by the *uniform-consensus()* function is employed as a termination protocol. We assume that every participant P_i has access to a local failure detector module FD_i that informs it of the list of participants that it currently suspects to have crashed. In the notations, $P_j \in FD_i$ means that participant P_i suspects participant P_j. *Task 1* implements the main protocol, while *Task 2* is used in order to ensure that if a correct participant receives the decision message, then all correct participants eventually receive this message. The main protocol operates in two steps as follows (*Task 1*):

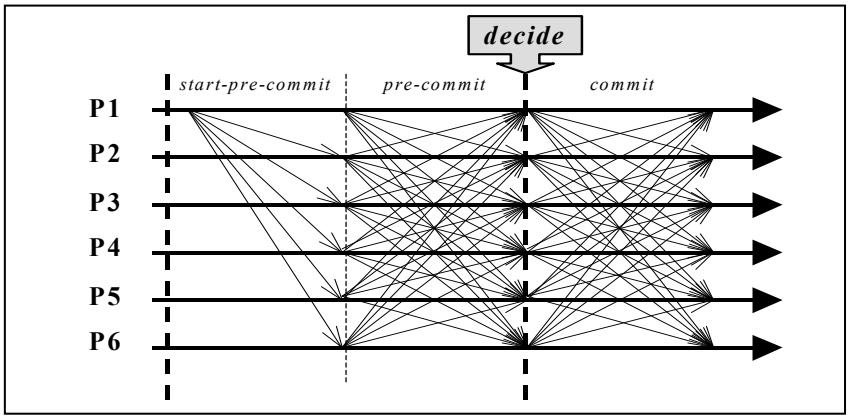

Figure 5.4: *The ANB-CLL protocol.*

During step1 (lines 1-10), if the coordinator's proposition is *abort* (line 1), then the coordinator sends an *abort* decision message to all (line 2), and decides *abort* (line 3); otherwise, the coordinator sends a *start-pre-commit* message to all (line 4). Each participant P_i waits until (i) it receives a *start-pre-commit* message from the coordinator, or (ii) it suspects the coordinator (line 5). In case (ii), P_i asks all other participants to start a uniform consensus protocol by sending a *start-consensus* message to all (line 7), then P_i starts a uniform consensus with *abort* as its initial value (line 8). When the uniform consensus protocol returns a *decision*, P_i decides accordingly (line 9); In case (i), P_i sends a *pre-commit* message to all (line 10), and proceeds to step 2 of the protocol.

During step 2 (lines 11-17), each participant P_i waits until (i) it receives a *pre-commit* message from all, or (ii) it receives a *start-consensus* message, or (iii) it suspects another participant (line 12). In case (i) P_i sends a *commit* decision message to all (line 16), and decides *commit* (line 17). In cases (ii) and (iii) (line 13), P_i starts a uniform consensus protocol with *commit* as its initial value (line 14). When the uniform consensus protocol returns a *decision*, P_i decides accordingly (line 15).

During Task 2 (lines 18-20), a participant P_i waits until it receives a *decision* message (line 18), sends the *decision* to all (line 19), and decides upon this *decision* (line 20).

```
function terminate()

// Task1
Only the coordinator executes:
1  If proposition = abort then
2   send (abort) to all participants;
3   return (abort);        // decide abort
4  send (start-pre-commit) to all participants;          // proposition = commit

// Every participant P_i executes:
5    wait until [received (start-pre-commit) from coordinator or coordinator ∈ FD_i];
6         if coordinator ∈ FD_i then
7              send (start-consensus) to all participants;
8              decision := uniform-consensus(abort);
9              return (decision);      // decide decision
10   send (pre-commit) to all participants;

11   for every participant P_j in the transaction:
12        wait until [received ((pre-commit) or (start-consensus)) from P_j or P_j ∈ FD_i];
13             if received (start-consensus) from P_j or P_j ∈ FD_i then
14                  decision := uniform-consensus(commit);
15                  return (decision);      // decide decision
16        send (commit) to all participants;
17        return (commit);     // decide commit

// Task2
18   wait until [received (decision) from any P_j];
19        send (decision) to all participants;
20        return (decision);      // decide decision
```

Figure 5.5: *The ANB-CLL protocol.*

5.4.5 Protocol Correctness

In this section, we show that our ANB-CLL protocol presented in Figure 5.5 is correct and non-blocking. This amounts to proving that it satisfies all of the five properties of the NB-WDAC problem.

Theorem 5.6. *ANB-CLL achieves the DAC-Uniform-Validity property.*

PROOF. Assume that no participant suspects the coordinator during the protocol execution. Since a participant can decide *commit* (resp., *abort)* in Task 2 only if some participant has decided *commit* (resp., *abort)* in Task 1, we only need to show that (i) if the coordinator's proposition is *commit*, then no participant can decide *abort* in Task 1, and (ii) if the coordinator's proposition is *abort*, then no participant can decide *commit* in Task 1.

Case (i): For contradiction, assume that a participant P_i decides *abort* in Task 1. In Task 1, a participant can decide *abort* only at lines 3, 9, and 15. Since no participant suspects the coordinator, then P_i could not have decided *abort* at line 9. To decide *abort* at line 3, P_i must be the coordinator of the protocol. For the coordinator to reach line 3, its proposition must be *abort*: a contradiction. To decide *abort* at line 15, P_i must have gotten *abort* as the outcome of the uniform consensus of line 14. By the *C-Uniform-Validity* property of uniform consensus, some participant P_j must have started uniform consensus with *abort* as its initial value at line 8. For P_j to reach line 8, P_j must have suspected the coordinator at line 5: a contradiction with the assumption that no participant suspects the coordinator. Therefore, no participant can decide *abort* in Task 1.

Case (ii): For contradiction, assume that a participant P_i decides *commit* in Task 1. In Task 1, P_i can decide *commit* only at lines 9, 15, or 17. Since no participant suspects the coordinator, then P_i could not have decided *commit* at line 9. Hence, P_i must have decided *commit* at lines 15 or 17. For P_i to reach lines 15 or 17, P_i must have received a *start-pre-commit* message from the coordinator (line 5). This means that the coordinator must have executed line 4. For the coordinator to

execute line 4, the coordinator's proposition must be *commit*: a contradiction. Thus, no participant can decide commit in Task 1.

Theorem 5.7. *ANB-CLL achieves the DAC-Uniform-Agreement property.*

PROOF. A participant can decide *commit* (resp. *abort*) in Task 2 only if some participant has decided *commit* (resp. *abort*) in Task 1. We show that no two participants can decide differently in Task 1. In Task 1, a participant can only decide at lines 3, 9, 15, and 17. We have to consider two cases: (i) the coordinator decides (*abort*) at line 3, or (ii) the coordinator does not decide at line 3.

Case (i): In this case, the coordinator does not execute line 4, and hence does not send *start-pre-commit* to all. Therefore, no participant decides at lines 15 or 17. Thus, every participant (that decides) decides at line 9 following the execution of the uniform consensus (of line 8) with *abort* as initial value. By the *C-Uniform-Validity* property of uniform consensus, every participant (that decides) decides *abort*.

Case (ii): There are two sub-cases to consider: (a) no participant suspects the coordinator during step 1, or (b) at least one participant P_i suspects the coordinator during step 1. In (b), P_i starts a uniform consensus with *abort* as its initial value (line 8), and thus, does not send a *pre-commit* message to all. This means that no participant can decide at line 17, since the *pre-commit* message of P_i is missing. Consequently, every participant (that decides) decides either at line 9 or at line 15 following the execution of the uniform consensus (started either at line 8 or at line 14). By the *C-Uniform-Agreement* property of the uniform consensus, no two participants decide differently. In (a), no uniform consensus is started (at line 8) with *abort* as initial value. By the *C-Uniform-Validity* condition of uniform consensus, no participant decides *abort* at line 15. Hence, every participant (that decides) decides *commit*, either at line 15 or at line 17.

Theorem 5.8. *ANB-CLL achieves the DAC-Non-Blocking property (assuming a failure detector of class $\diamond S$, and a majority of correct participants).*

PROOF. We consider two cases: (i) at least one correct participant does not execute step 2, and (ii) all correct participants execute step 2.

Case (i): Assume that the coordinator crashes. By the *strong completeness* property of $\diamond S$, every correct participant eventually suspects the coordinator. If, however, the coordinator is correct, then it eventually sends either (a) a *start-pre-commit* message to all (line 4) or (b) an *abort* decision message to all (line 2). In (a), if the coordinator sends a *start-pre-commit* message to all (line 4), then every correct participant executing the wait statement at line 5 eventually receives this message (by the reliable communication assumption). Therefore, if a participant P_i does not execute step 2, then P_i must have suspected the coordinator in step 1, in which case, P_i sends a *start-consensus message* to all participants (line 7) and starts uniform consensus (line 8). Since P_i does not execute line 10, correct participants that have reached step 2 do not receive the *pre-commit* of P_i. Hence, either they receive the *start-consensus* message of P_i (reliable communication), or they suspect another participant. In both cases, every correct participant in step 2 eventually starts uniform consensus (line14). Since we assume a majority of correct participants, and by the *C-Non-Blocking* property of uniform consensus, every correct participant eventually decides. In (b), if the coordinator sends an *abort* decision message to all (line 2), then every correct participant that has not decided yet eventually receives this message (reliable communication), and decides accordingly (at line 20 of Task 2).

Case (ii): In this case, either some correct participant receives *pre-commit* from all, or no correct participant receives *pre-commit* from all. If some correct participant in step 2 receives *pre-commit* from all, then this participant sends *commit* to all (line 16), and decides *commit* (line 17). Hence, every correct participant that has not decided yet eventually receives the *commit* decision message (by the reliable communication assumption) and decides commit (at line 20 of Task 2). The case where no correct participant receives *pre-commit* from all is subtler: all correct participants execute step 2 means that all correct participants sent their *pre-commit* to all. If all participants are correct, then all correct participants receive *pre-commit* from all due to reliable communication: a contradiction. It follows that some participants are not correct. By the strong completeness property of $\diamond S$, all correct participants eventually suspect another participant (line 12). Thus, every correct participant in step 2 eventually starts uniform consensus (line 14). Since we assume a majority of correct participants, then a majority of correct participants eventually

start uniform consensus. Again, by the *C-Non-Blocking* property of uniform consensus, every correct participant eventually decides.

Theorem 5.9. *ANB-CLL achieves the DAC-Uniform-Integrity property.*

PROOF. From the structure of the protocol, it is clear that every participant decides at most once (either in Task 1 or in Task 2).

Theorem 5.10. *ANB-CLL achieves the DAC-Termination property.*

PROOF. To show that *DAC-Termination* is satisfied, we must consider participants' recovery. Given that ANB-CLL exploits the same recovery procedure as CLL (and NB-CLL), the proof remains the same for both protocols (cf. Section 3.5.3).

5.4.6 Performance Evaluation

In this section, we examine the cost for non-blocking under the ANB-CLL protocol, and compare it with existing non-blocking protocols proposed in the same context, namely DNB-AC and MD3PC. Figure 5.6 summarizes the performances of the protocols in terms of latency and message complexity needed to commit a transaction. We denote by n the number of participants in the transaction and by f the number of crash failures to be tolerated, where $f < \lceil n/2 \rceil$.

By exploiting the 1PC approach to distributed transaction commit, ANB-CLL reduces the time complexity of both DNB-AC and MD3PC from 3 communication steps to 2, thus reducing transaction response times.

Regarding message complexity, we distinguish two cases: (1) with a broadcast network, and (2) without a broadcast network. In case (1), with 6 participants ($n = 6$) and a resiliency rate of 1 ($f = 1$), DNB-AC requires 114 messages, whereas both MD3PC and ANB-CLL require 78 messages. In case (2), DNB-AC requires 19 messages, MD3PC requires 16 messages, and ANB-CLL requires 13 messages. To illustrate further, assume now that $n = 12$ and $f = 2$. In case (1), DNB-AC requires 444 messages, MD3PC requires 276 messages, while ANB-CLL requires 300 messages. In case (2), DNB-AC requires 37 messages, MD3PC requires 30 messages, and ANB-CLL requires 25 messages.

	Message Complexity		Latency	
	point-to-point network	*broadcast network*	Time complexity	Log Complexity
DNB-AC	$n + 3n^2$	$3n + 1$	3	--
MD3PC	$n(4f + 3) + n^2$	$2(n + f + 1)$	3	--
ANB-CLL	$n + 2n^2$	$2n + 1$	2	$n + 1$

Figure 5.6: *The cost of transaction commit under DNB-AC, MD3PC, and ANB-CLL.*

To summarize, we note that, independently of the number of participants in a transaction, ANB-CLL reduces the message complexity of DNB-AC under both types of networks, and that of MD3PC when a broadcast network is used. In case of a point-to-point network, if the number of participants exceeds 6 (i.e., $n > 6$), more messages need to be exchanged in ANB-CLL than in MD3PC. This is rather not surprising given that in MD3PC, the sub-protocol required for non-blocking is executed only by a subset of the participants in the transaction, and the cardinality of this subset depends on the number of failures to be tolerated. Although in real-world transactional applications the number of participants rarely exceeds 6, we can perfectly apply this optimization to ANB-CLL so as to trade the resiliency of the protocol with the number of messages exchanged, thus making its message complexity always below that of MD3PC: in this case, $(n+1) + (2f+1) + n^2$ (resp. $n + 2f + 2$) messages would be needed assuming a point-to-point network (resp. a broadcast network), making a total of 209 (resp. 18) messages, with $n = 12$ and $f = 2$. This gives ANB-CLL the best overall performances among the three discussed protocols.

Like DNB-AC and MD3PC, our ANB-CLL protocol achieves non-blocking in asynchronous systems assuming reliable communication, any unreliable failure detector of class $\diamond S$, and a majority of correct participants. If these assumptions are not satisfied, our protocol might block, but never leads two participants to decide on different outcomes.

5.5 Discussion

The work presented in this chapter originated from the observation that, in today's transactional systems and applications, high performance and fault-tolerance are crucial requirements of equal importance.

Based on this observation, and given the high efficiency of the 1PC approach to distributed transaction commit, we were prompted to investigate fault-tolerant solutions to the Dictatorial Atomic Commitment problem. This led us to propose two non-blocking extensions to CLL, our 1PC variation, which provide transaction liveness guarantees under the two extremes of a spectrum of possible system models, namely synchronous and asynchronous systems. The resulting protocols, which we called NB-CLL and ANB-CLL, blend the efficiency of 1PC with fault-tolerance. The importance of this work is further emphasized by the fact that, compared to 2PC, 1PC increases the probability to blocking of the participating sites in case of failures.

The advantages of NB-CLL and ANB-CLL over other non-blocking protocols proposed in the literature are not only performance issues. By combining the 1PC approach with our CLL's recovery mechanism, our protocols are able to cope with existing systems without violating their autonomy — be they or not 2PC compliant.

Furthermore, by adapting Chandra & Toueg's consensus protocols [Cha93, CT96] to the transactional context, and based on the recovery semantics of the distributed commit problem, ANB-CLL achieves non-blocking in asynchronous systems assuming a *crash-recovery* failure model. To the best of our knowledge, it is the first time that fault-tolerant solutions to the distributed commit problem have been devised for asynchronous systems in which processes may crash and later recover. With all these features, ANB-CLL is able to meet the fundamental requirements of today's real-world transactional systems and applications.

Part III

Pragmatic Implementation

Chapter 6

The ANB-CLL Prototype

In this chapter, we show how to put our theoretical results into practice by presenting a way by which 1PC can be exploited in current transactional standards and products, initially designed with 2PC in mind. To do so, we first give an overview of well-established TP standards promoted by ISO, X/Open, and OMG, which have gained widespread acceptance and commercial product support. We then show how our ANB-CLL protocol, discussed in the previous chapter, can be smoothly integrated into these standards through a prototype design and implementation achieved in the context of this thesis.

6.1 Transactional Standards

This section recalls some background related to the ISO OSI-TP protocol, X/Open DTP model, and OMG's OTS service.

6.1.1 The ISO OSI-TP Protocol

OSI-TP (Open Systems Interconnection - Transaction Processing) [ISO92a] is a transactional protocol defined by ISO (International Standardization Organization), which guarantees *interoperability* between different transactional components (e.g., TP monitors) involved in the commitment of a distributed transaction. More precisely, OSI-TP defines (i) a standard communication protocol for establishing and managing dialogs between participants in a transaction, (ii) a standard two-phase commit (2PC) protocol, and (iii) a standard failure management and recovery protocol.

As far as atomic commitment is concerned, OSI-TP integrates several optimizations of basic 2PC, namely the Presumed Abort (PrA) (cf. Section 2.4.1), Read-Only (cf. Section 2.4.4), and One Phase Commit optimizations. We caution the reader that the latter is not to be confused with our 1PC concept as it has totally different semantics — it is rather intended to optimize the cost of commit processing in case of *mono-site* transactions, i.e., when there is only one participant in the transaction.

Given that 2PC is a blocking protocol, a *heuristic decision* concept has been also adopted in order to resolve blocking situations that may arise in case of failures. More precisely, if a coordinator crash occurs, an uncertain participant can unilaterally *commit* or *abort* the transaction rather than waiting for the coordinator to recover. Upon recovery, if the coordinator's final decision contradicts the participant's heuristic decision, a manual procedure is launched to reestablish a global consistent state. Thus, non-blocking is obtained at the expense of data consistency.

6.1.2 The X/Open DTP Model

The DTP (Distributed Transaction Processing) model [X/Open93] is a transactional standard promoted by X/Open, which aims at providing standard interfaces between transactional components so as to make them *portable*. This model distinguishes four software entities that participate in the execution of a transaction: (i) an *Application Program* (AP) is an arbitrary program that implements the desired function of the end-user application, and accesses shared resources within the scope of a transaction, (ii) a *Resource Manager* (RM), (e.g., a Database Management System, or simply DBMS), manages shared resources and guarantees the consistency of data it is in charge of, (iii) a *Transaction Manager* (TM) (e.g., a TP-Monitor) coordinates atomic transaction completion at all RMs accessed by a transaction, and manages failure recovery, and (iv) a *Communication Resource Manager* (CRM) facilitates interoperability between different instances of the DTP model by managing communication between distributed and potentially heterogeneous TMs located in different domains, and provides portable APIs for DTP communication between several APs.

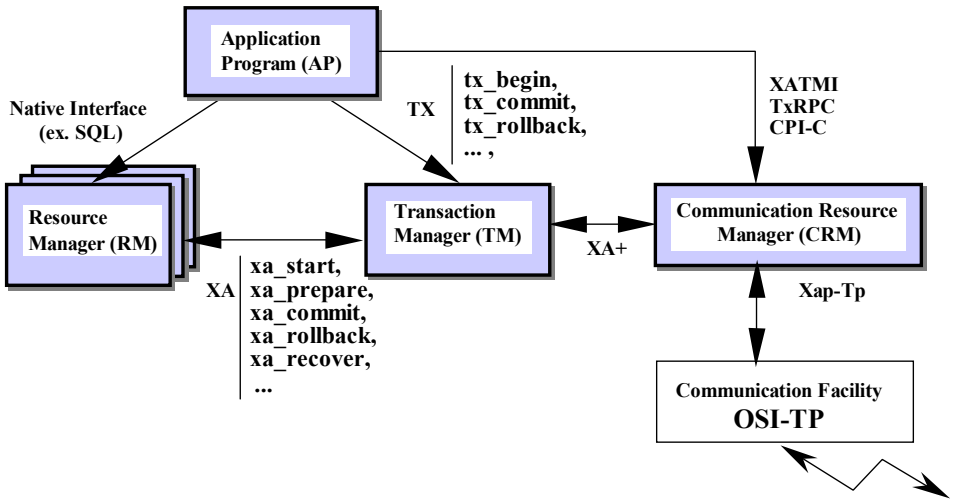

Figure 6.1: *X/Open DTP model.*

Figure 6.1 illustrates the functional components of a local instance of a DTP system. Typically, an AP accesses the TM through the TX interface in order to begin/commit/abort a transaction, and accesses RMs through their native interface (e.g., SQL). When the AP requests the TM to commit a transaction, the latter acts as the coordinator of the commit protocol during which it directs the different participating RMs for a *commit* or an *abort* through their XA interface. As defined in OSI-TP, the commit protocol adopted in the X/Open DTP model is the PrA 2PC protocol, together with the Read-Only and One Phase Commit optimizations.

In case several distributed (possibly) heterogeneous TMs are involved in the execution of the same transaction, they communicate through their respective CRMs using the OSI-TP protocols in order to exchange DTP information and application data (Figure 6.1). Thus, X/Open DTP ensures the *portability* of transactional components while OSI-TP ensures their *interoperability*.

6.1.3 The OMG Object Transaction Service

Oriented towards the object world, OMG (Object Management Group) has specified a transactional standard, named OTS (Object Transaction Service) [OMG00a], based on the CORBA architecture ratified by the members of OMG [OMG00b].

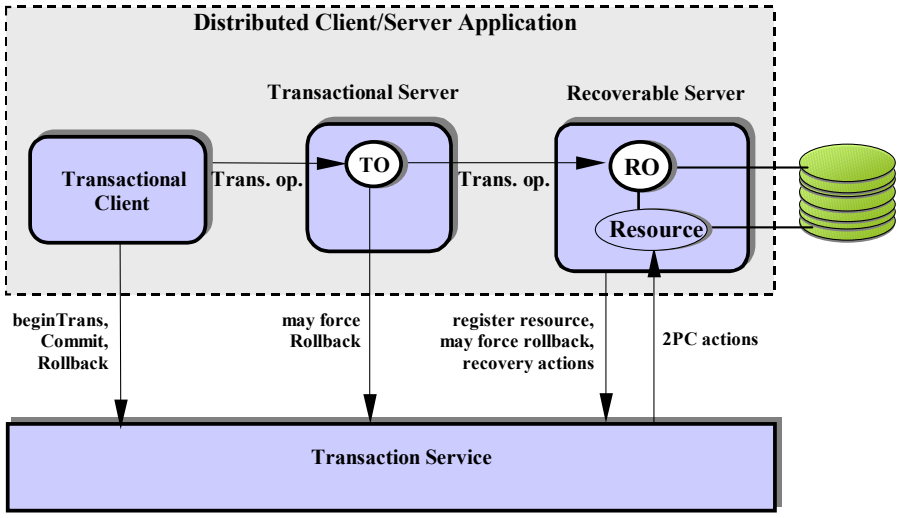

Figure 6.2. *OMG's OTS architecture.*

Simply stated, CORBA provides a distributed object-oriented infrastructure that allows objects to communicate across boundaries such as the network, the specific language in which they were written or the platform on which they are deployed. The communication heart of the CORBA architecture is the Object Request Broker (ORB) that acts as the object bus over which objects transparently interact with other remote objects. OTS brings the notion of distributed transactions to the CORBA world.

OTS Architecture

As illustrated in Figure 6.2, the CORBA OTS model distinguishes six main entities that participate in the execution of a transaction: (i) a *Transactional Client* (TC) is an arbitrary program that invokes operations on transactional objects within the scope of a transaction, (ii) a *Transactional Object* (TO) is an application object whose behavior is affected by being invoked within the scope of a transaction, (iii) a *Recoverable Object* (RO) is an application object that directly manages persistent data whose state is subject to change during the course of a transaction, and thus must participate in the 2PC protocol defined by OTS[22], (iv) a *Transactional Server* is a collection of one or more transactional (but not recoverable) objects, (v) a *Recoverable Server* is a collection of

[22] In accordance with OSI-TP and X/Open DTP, the commit protocol defined in OTS is the PrA variation of basic 2PC, together with the Read-Only and One Phase Commit optimizations.

objects, at least one of which is recoverable, and (vi) the *Transaction Service* coordinates all the transactions in the system, and drives the 2PC protocol.

A Recoverable Object participates in the 2PC protocol by registering an object called *Resource* with the Transaction Service. The Resource object implements the 2PC protocol as a participant on behalf of the Recoverable Object in order to update the Recoverable Object's data resources in accordance with the transaction outcome. At transaction end, the Transaction Service drives the 2PC protocol by issuing requests to all the resources registered for the transaction.

Note that even though a Recoverable Object is by definition a Transactional Object, an object can be Transactional but not Recoverable, in which case it does not directly manage persistent data, but rather, it invokes operations on some other Recoverable Object(s). Consequently, Transactional objects *that are not Recoverable* do not participate in the 2PC protocol; however, they may force the rollback of the transaction.

Principal OTS Interfaces

In OTS, a transaction is managed by a set of CORBA objects, each having a standard interface defined in terms of the OMG's *Interface Definition Language* (IDL). Figure 6.3 illustrates the key interfaces defined in OTS together with the major components using them. These interfaces are discussed below:

- ***Current interface*:** provides application objects with a transparent access to the Transaction Service. It can be used to *begin*, *commit*, or *rollback* a transaction, and to get information about the current transaction.
- ***Transaction Factory interface*:** allows the Transactional Client to begin a transaction.
- ***Control interface*:** can be viewed as the handle to the transaction. More precisely, it provides access to two other interfaces that control the transaction life cycle, namely the *Coordinator* and the *Terminator* interfaces, thus enabling the application to interact directly with the Transaction Service objects.
- ***Coordinator interface*:** provides operations used by participants in a transaction, and supports mechanisms to coordinate transaction termination at these participants.
- ***Terminator interface*:** provides operations to *commit* or *rollback* a transaction.

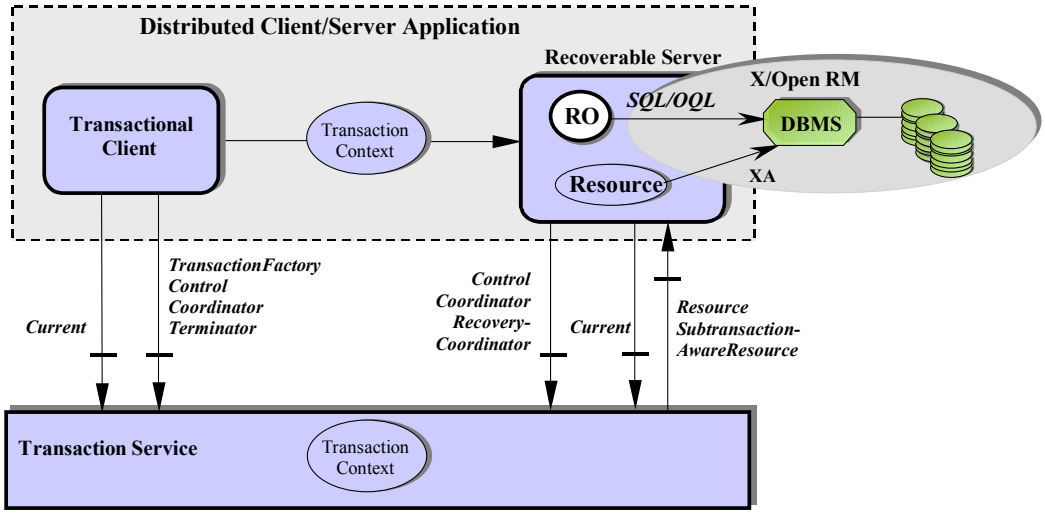

Figure 6.3. *Key interfaces in OTS.*

- ***Resource interface*:** defines the operations invoked by the Transaction Service to complete a transaction on a resource following the 2PC protocol. This interface can be used to wrap non-CORBA resources to the CORBA domain so that they can participate in a CORBA transaction.
- ***Recovery Coordinator interface*:** is used by Recoverable Objects to drive the recovery process in case of failures.
- ***Subtransaction Aware Resource interface:*** is a specialization of the *Resource* interface, used by Recoverable Objects that support the nested transaction behavior.

It is very important to note that one of the major goals of the OTS specification is to allow legacy TP-based systems to participate in an OTS transaction. In particular, OTS is designed to interact with X/Open DTP-compliant Resource Managers, or simply RMs (Figure 6.3). This actually means that OTS Recoverable Objects can use X/Open RMs interfaces (e.g., SQL) and access the data resources they manage within the scope of an OTS transaction. In this case, the registered *Resource* object represents the accessed RM as a participant in the transaction completion. Recall that X/Open RMs can participate in a distributed transaction by allowing their 2PC protocol to be controlled via the XA interface (cf. Section 6.2). Therefore, to complete a transaction, the Transaction Service drives the commit protocol by issuing 2PC requests on the registered *Resource*, while the *Resource* drives the RM through its XA interface as we further detail in the following.

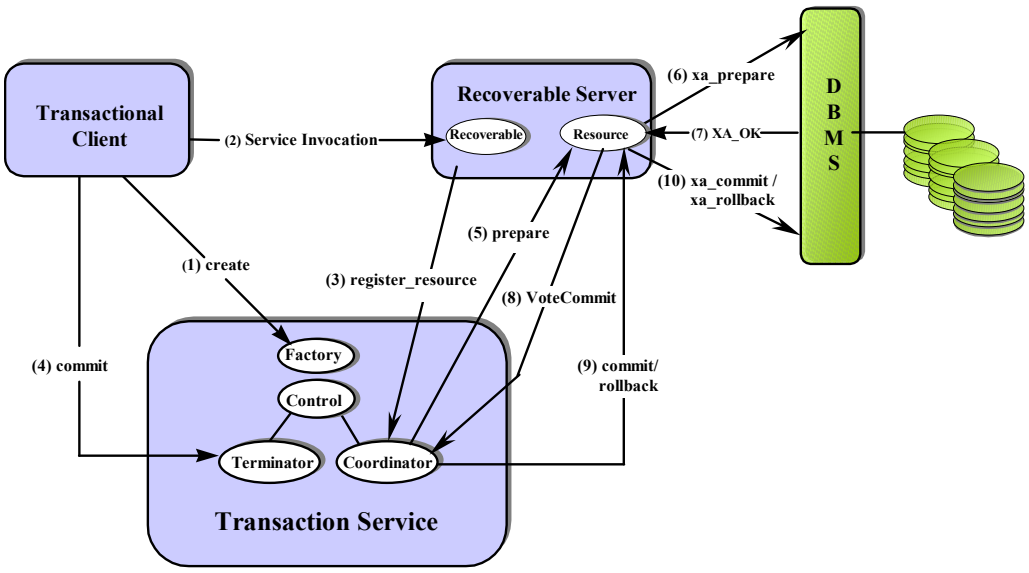

Figure 6.4: *OTS execution flows using direct transaction management.*

Typical Usage

In OTS, client applications manage their transactions either *directly* or *indirectly*.

- With *direct* transaction management, the client application directly accesses and manipulates the Transaction Service objects that represent the transaction (i.e., *Transaction Factory*, *Control*, *Terminator*, *Coordinator*, etc.). Figure 6.4 illustrates a typical OTS transaction execution using the direct mode. The Transactional Client starts a transaction using a *Transaction Factory* object. A *Control* object is returned, which provides access to a *Terminator* and a *Coordinator*. Then, the client starts sending requests to the Recoverable Server, and includes in each of its requests the transaction context[23], which can be obtained from the *Coordinator* object. On receipt of a service request, the Recoverable Server registers a *Resource* object with the *Coordinator*. At transaction end, the client uses the *Terminator* object to *commit* or *rollback* the transaction. On a *commit* request, the Transaction Service starts the 2PC protocol by issuing requests to all the *Resources* registered with the *Coordinator*.

[23] An OTS transaction context generally contains the object reference to the transaction Coordinator together with a unique global transaction identifier.

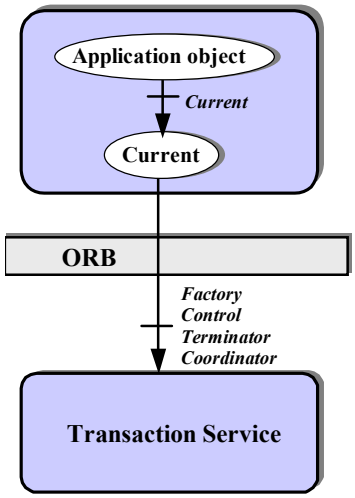

Figure 6.5: *Indirect transaction management mode.*

- With *indirect* transaction management, the set of OTS interfaces are hidden by the *Current* pseudo object, which provides a fully transparent access to OTS. Figure 6.5 illustrates the role of the *Current* object and its relation with application objects and the Transaction Service objects. Requests from the application object to the *Current* pseudo object are local requests. The *Current* interacts with the Transaction Service objects through the ORB as an application object using direct transaction management mode. Thus, the *Current* can be viewed as a high level API that hides the location of the Transaction Service and the set of its interfaces.

6.1.4 OTS and DTP Compared

OTS can be seen as an object redefinition of the X/Open DTP model. It brings the transaction paradigm and the object paradigm together, thus promoting *reliable*, *modular*, *reusable* and *evolutionary* object-based software components. Most importantly, OTS has been designed to be compatible with well-established transactional standards, thus enabling the integration and interoperability of legacy TP based systems with the CORBA domain. In particular, OTS is fully compatible with X/Open DTP-compliant software, which allows a single (X/Open or OTS) transaction to be shared by both object and procedural code.

Fully based on the CORBA architecture, inter-component communications in OTS are all in the form of object requests sent via the ORB, which enables access and location transparency of remote objects. This is compared to the X/Open DTP model where a Communication Resource Manager (CRM) is required to process transactions that are distributed over several TMs.

6.2 ANB-CLL in Standard Platforms

Given that the transactional models presented in the previous section are well-established TP standards that have gained widespread acceptance and commercial product support, it is important to show how our ANB-CLL protocol, described in the previous chapter (Section 5.4.4), can be exploited in an OTS/DTP environment.

To do so, we first show how our (blocking) CLL protocol (cf. Section 3.5) can be embedded within a fully OTS-compliant Transaction Service, named MAAO-OTS [LSG98], while maintaining the interoperability of DTP-compliant systems with the CORBA domain. We then describe how to achieve non-blocking by exploiting a CORBA compliant service, called OGS [Fel98], which defines an object-oriented framework of CORBA components for reliable distributed systems.

6.2.1 Prototype Context

The ANB-CLL prototype has been performed in the context of OpenDREAMS-II[24], an ESPRIT project financed by the European Union (December 1997 -- May 2000). OpenDREAMS-II (henceforth called "OD-II") aims at designing and building a CORBA compliant platform dedicated to industrial Supervision and Control Systems (SCS). The OD-II platform is augmented with several components and services specifically tailored to answer SCS requirements, including a Transaction Service designed and implemented by the PR*i*SM laboratory [ABG98].

The project platform is experimented and validated through two industrial SCS applications, namely a Condition Monitoring and Diagnostics of Thermal Power Plants

[24] OpenDREAMS is the acronym for "Open Distributed Reliable Environment, Architecture & Middleware for Supervision".

application, as well as an Advanced Surface Movement Guidance & Control Systems (A-SMGCS) application for managing all moving vehicles in an airport environment. Both applications showed the effectiveness of our protocol in meeting SCS requirements in terms of performance, fault-tolerance, and compliance with commercial transactional systems.

6.2.2 Major Objectives

When defining the overall project goal, we have set out the following major objectives for our ANB-CLL prototype:

- To show the applicability of the 1PC idea in general, and our protocol in particular, to real-world transactional systems and standards.
- To enable application portability from the OD-II Transaction Service to other OTS implementations by following the standard OTS interfaces defined by OMG.
- To enable the integration of X/Open DTP-compliant transactional systems in the OD-II Transaction Service by directing them through their standard XA interface.

6.2.3 Integrating CLL into OTS

In this section, we show how the (blocking) CLL protocol can be embedded within a fully OTS compliant Transaction Service, named *MAAO-OTS* [LSG98], developed by the *TRANSREP* project members headed by Simone Sédillot at *INRIA* [25]. The CLL prototype components have been fully designed, and implemented in C++ using Orbix 2.3 MT [ION97], a commercial CORBA implementation.

Transactional Client

In the OD-II Transaction Service, the support of the CLL protocol is totally *transparent* to the client application. More precisely, a client of the OD-II Transaction Service still accesses the standard OTS interfaces as defined by OMG to begin (resp. commit) its transaction by calling the standard *begin()* (resp. *commit()*) operation on the *Current*

[25] INRIA is the acronym for “Institut National de Recherche en Informatique et Automatique”.

object (*indirect* mode), or the *Factory* (resp. *Terminator*) object (*direct* mode). The call to *commit()* on either object launches the CLL protocol implemented by the OD-II Transaction Service, and commits the transaction in a single phase on the participating resources.

Transaction Service

The integration of the CLL protocol within the OD-II Transaction Service has been realized thanks to the collaboration of the TRANSREP project members at INRIA. This integration consisted in modifying the MAAO-OTS coordinator automaton so as to follow the 1PC approach rather than the traditional 2PC approach. The necessary modification is rather straightforward, and is achieved by simply having the coordinator ask the registered *Resource* objects to commit the transaction without first asking them to prepare.

It is very important to note that eliminating the voting phase from the commit protocol does not require any modification/extension to the *Resource* interface as defined by OMG. Instead, we exploit the standard *commit_one_phase()* operation (traditionally offered by the *Resource* interface and employed by the transaction coordinator in case of mono-site transactions) for our 1PC purpose. Clearly, a call to *commit_one_phase()* on each participating resource is mapped to a call to *xa_commit(TMONEPHASE)* on the corresponding X/Open DTP-compliant RM.

Recoverable Server

Recall from Chapter 3 (Section 3.5.2) that the concept of *Agent* has been associated with each transactional system (typically, RMs) participating in the CLL protocol. The role of the *Agent* is to determine the exact state (i.e., *committed* or *aborted*) of every transaction branch for which its local site did not acknowledge the *commit* decision due to a failure. This is important in order to identify those branches that need to be locally re-executed.

In the OD-II Transaction Service, we have integrated the *Agent* role within the *Resource* object. Obviously, this is the most natural and straightforward way to do since the *Resource* object is the entity that acts as intermediary between the Transaction Service and the underlying participating RMs.

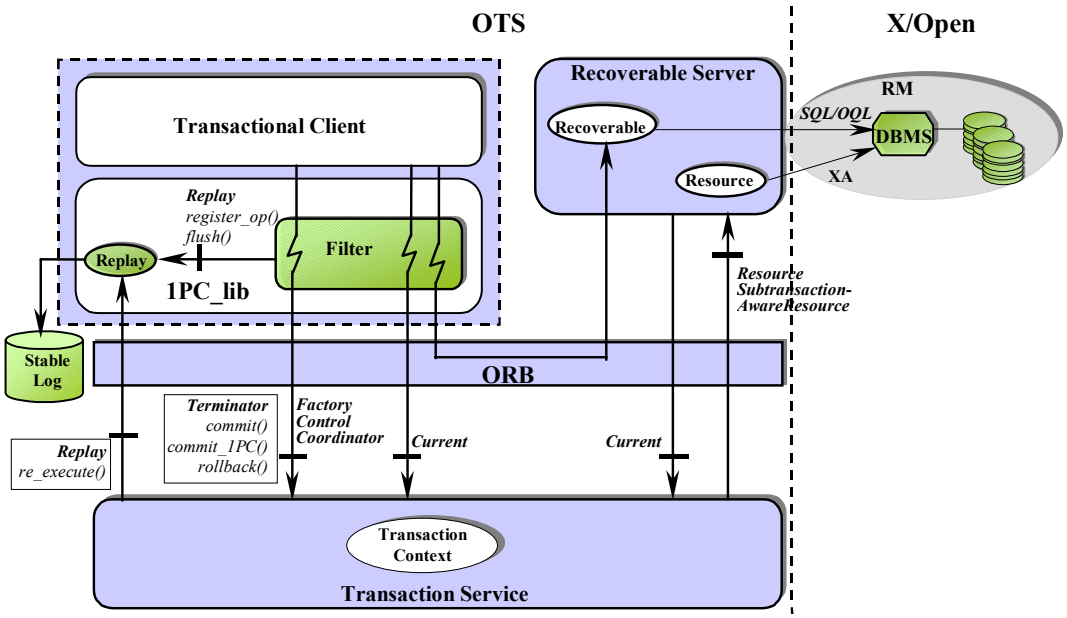

Figure 6.6: *1PC_lib in the OD-II Transaction Service architecture.*

Achieving Commit-resiliency

Recall that to overcome the need for *on-line commit-resiliency* at the participants while preserving their autonomy, the coordinator of CLL guarantees the *commit-resiliency* property of transactions by maintaining in its log the list of operations invoked within the scope of a transaction. In addition, the CLL coordinator forces its log on stable storage before sending the *commit* decision to the different participants. In case a participant crashes during the CLL protocol execution, the coordinator re-executes the transaction branch on the failed participant.

In an OTS architecture, the difficulty in meeting this requirement lies in the fact that a Transactional Client sends its service requests directly to Recoverable Objects. Thus, at commit time, the *Coordinator* object has no knowledge of the list of requests invoked within the scope of a transaction.

To deal with this problem, our solution consists in keeping the list of a transaction's requests in a log maintained on the Transactional Client side. This log is kept transparent to the client application by means of a library, called *1PC_lib*, dedicated to CLL's specific mechanisms and to which the client application should be linked. More precisely,

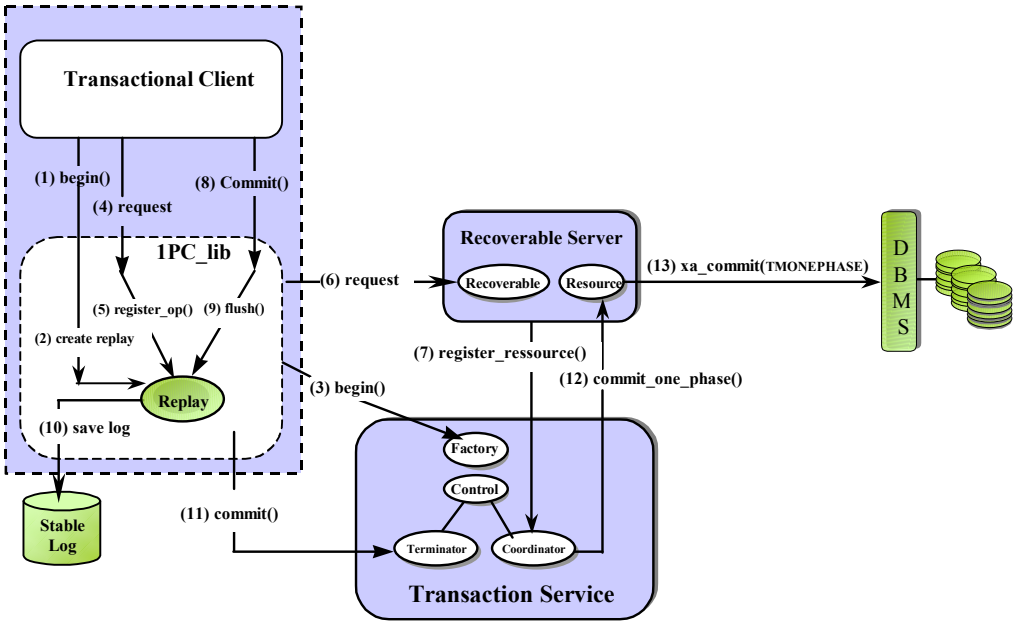

Figure 6.7: *OD-II Transaction Service execution flows.*

1PC_lib manages the requests' log via a new object that we introduce, called *Replay*. The *Replay* interface defines operations that allow to (i) write the transaction's requests on the log (*register_op()* operation), (ii) force the log on stable storage (*flush()* operation), and (iii) re-execute the requests of a transaction branch in the event of a participant crash during the CLL protocol execution (re*_execute()* operation).

Detailed Description

1PC_lib is implemented using Orbix *Per-Process Filters*[26]. Per-Process filters monitor all incoming and outgoing operation and attribute requests to and from an address space. Figure 6.6 illustrates the role of *1PC_lib* in the OD-II Transaction Service architecture. A typical transaction execution is described in Figure 6.7.

When a client application begins a transaction by calling *begin()* on the *Factory* object (or the *Current* object), the client filter associates a *Replay* object with the new transaction. During the transaction, the Transactional Client invokes service requests on

[26] The filter concept has been first introduced in IONA's Orbix ORB, but has been since normalized in CORBA 2.2 under the *Interceptors* concept.

Recoverable Objects. The client filter intercepts each of these requests, registers the request in the log by calling *register_op()* on the *Replay* object, and continues the call normally.

When the Transactional Client calls *commit()* on the transaction *Terminator* object (or the *Current* object), the client filter intercepts the call to *commit(),* force-writes the log (i.e., the transaction's requests) on stable storage by calling *flush()* on the *Replay*, and continues the call normally. The call to *commit()* launches the CLL protocol and commits the transaction in a single phase, while ensuring the transaction *commit-resiliency* property at the client side.

In case a participant crashes during the CLL protocol execution, the *Coordinator* "replays" the failed transaction branch by calling the *re_execute()* operation on the transaction's *Replay* object with the corresponding *Resource* object reference as a parameter. Note that the *Replay* object reference can be made available to the Transaction Service by having it piggybacked to the *commit()* request message by the client filter, and extracted by a receiving filter on the Transaction Service side.

6.2.4 Achieving Non-Blocking

This section briefly presents the design of a non-blocking extension to our CLL prototype, following the ANB-CLL protocol described in Section 5.4.4. The solution we propose exploits some of the facilities provided by a CORBA *Object Group Service* (OGS) [Fel98], designed and implemented at the *Operating Systems Laboratory* (LSE) directed by Professor André Schiper at the *Swiss Federal Institute of Technology* (EPFL).

Roughly, OGS provides object group support for CORBA environments by combining several distinct CORBA services, each providing a separate facility and can be exploited in isolation. Of particular importance for our purpose, an *Object Monitoring Service* that provides a distributed failure detection mechanism based on Chandra & Toueg's model of unreliable failure detectors [ChT96], and an *Object Consensus Service* that allows several CORBA objects to solve the Consensus problem based on Chandra & Toueg's Consensus algorithm, and using a failure detector of class $\diamond S$ [ChT96].

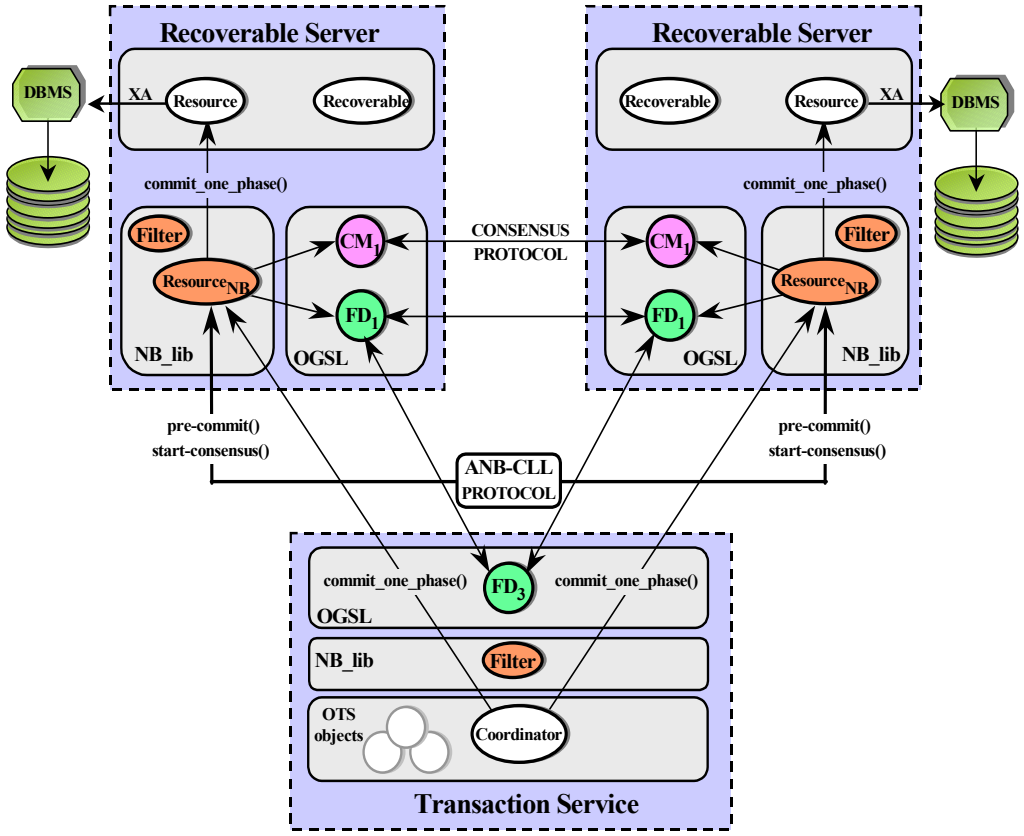

Figure 6.8: *Non-Blocking components of the OD-II Transaction Service.*

Components and Interactions

The non-blocking extension we propose is totally encapsulated within a library, called *NB_lib*, to which the Recoverable Server and the Transaction Service are linked. *NB_lib* manages all the non-blocking mechanisms introduced by ANB-CLL by defining new components, and by exploiting some of the services provided by OGS's library (*OGSL*). These mechanisms include a *pre-commit* phase, a Uniform Consensus algorithm, and a failure detection mechanism. Figure 6.8 presents a simplified high-level view of the non-blocking extension components.

On the Recoverable Server side, *NB_lib* introduces a new object, called $Resource_{NB}$, which acts as intermediary between the transaction *Coordinator* and the Recoverable Object's *Resource*, and implements the ANB-CLL protocol (cf. Figure 5.5) as a

participant on behalf of the Recoverable Object. From the transaction *Coordinator* viewpoint, the $Resource_{NB}$ object becomes the actual participant in transaction completion. To achieve its role, the $Resource_{NB}$ interface extends the standard *Resource* interface by defining, in addition to the *commit-one-phase()* operation, new operations required for non-blocking, namely *pre-commit(),* and *start-consensus().*

To deal with the failure detection problem, *NB_lib* exploits OGS's Object Monitoring Service by creating a *failure detector* object (*FD*) at the Recoverable Server, and the Transaction Service. Each local *FD* object monitors a subset of the processes in the system (roughly, by communicating with *FDs* local to these processes), and maintains a list of those processes that it currently suspects to have crashed. Given that in ANB-CLL, failure suspicions are handled within a Uniform Consensus protocol, *NB_lib* makes use of OGS's Object Consensus Service by creating a *consensus manager* object (*CM*) at each Recoverable Server. *CM* objects implement the consensus protocol and reach agreement with each other.

In this context, a $Resource_{NB}$ object acts as a client of its co-located *FD* and *CM* in order to get information about failure suspicions of other participating $Resource_{NB}$ objects, and to reach a consistent decision through the execution of a consensus protocol.

Typical Execution

When the Recoverable Object calls the *register_resource()* operation on the transaction *Coordinator* with a *Resource* object reference as a parameter, *NB_lib* intercepts the call by means of an *Interceptor* (or *Filter*), creates a new $Resource_{NB}$ object, and registers it with the *Coordinator* by modifying the value of the operation parameter to include the $Resource_{NB}$ object reference instead of that of the Recoverable Object's *Resource*.

To commit the transaction, the *Coordinator* performs the *commit-one-phase()* operation on every registered $Resource_{NB}$. This call initiates ANB-CLL's *pre-commit* phase between the different participating $Resource_{NB}$ objects through their respective *pre-commit()* operation. In the absence of failure suspicions, a *commit* decision is reached, in which case, the $Resource_{NB}$ performs *commit-one-phase()* on the Recoverable Object's *Resource*. In case of failure suspicions, the $Resource_{NB}$ asks the

CM object to start a uniform consensus protocol, and decides on the transaction (in a non-blocking way) according to the uniform consensus protocol outcome. Finally, note that if a failure suspicion occurs during the *pre-commit* phase, the $Resource_{NB}$ needs also to perform *start-consensus()* on the other participating $Resource_{NB}$ objects, as defined in ANB-CLL.

6.3 Discussion

In this chapter, we studied the integration of our ANB-CLL protocol into well-established TP standards that have gained widespread acceptance and commercial product support. Our primary objective here was to show the applicability of the 1PC concept in general, and our ANB-CLL protocol in particular, to real-world transactional systems and standards, namely OMG's OTS [OMG00a] and X/Open DTP [X/Open93].

This integration has been achieved following the same modular approach by which ANB-CLL has been designed. This consisted first in embedding the basic 1PC protocol (cf. Section 3.2) within an OTS architecture, and then encapsulating all CLL's specific mechanisms on the one hand and non-blocking facilities on the other within two separate libraries, named *1PC_lib* and *NB_lib*, respectively.

Our CLL prototype has been implemented in C++ using Orbix 2.3MT [ION97] based on a fully OTS-compliant transaction service, named MAAO-OTS [LSG98]. This enabled us to prove the practical validity of the 1PC concept, and to show the compatibility of our protocol with existing transactional standards and commercial database systems. As far as fault-tolerance is concerned, our non-blocking solution has been fully designed following a CORBA-compliant approach, but has not yet been implemented and integrated to the prototype due to timing and organizational constraints related to the OD-II project. It would be thus important to complete the present work, and study the cost for non-blocking through an actual implementation of the proposed solution in the context of real-world transactional systems based on a CORBA architecture.

Some issues related to the CLL prototype remain open for further investigations, notably concerning performance measurements. Although the performance gain of 1PC

over 2PC is obvious, it would be important to quantify this gain not only in terms of message and log complexities, but also in terms of *overall* transaction processing metrics, such as transaction *(peak) throughput* or transaction *(mean) response time*.

Chapter 7

Conclusion

Over the past two decades, distributed systems have become the norm for the organization of computing facilities. From common daily life activities to mission critical computing industries, everything shows evidence that we depend more and more heavily on distributed systems and applications, making the *reliability* of these more critical than it has ever been before.

Originated from the field of databases, the *transaction* abstraction has been widely acknowledged as the basic building block by which distributed systems and applications can be reliably structured and implemented. Reliability guarantees are provided despite concurrency and failures through transaction *ACIDity* (i.e., *atomicity*, *consistency*, *isolation*, and *durability*), where atomicity is ensured through an *atomic commitment protocol*, enabling a distributed agreement to be reached among participating processes concerning the faith of the transaction. Given their great impact on distributed transaction processing, a plethora of atomic commitment protocols has been proposed. These protocols, however, usually compel a trade-off between *high-performance and fault-tolerance* (i.e., *non-blocking*), making them inadequate for many of today's distributed systems and applications in which it becomes hardly acceptable to sacrifice one requirement for the other.

In this thesis, we have considered this issue through the discussion of the details involved in the design of a distributed commit protocol that reconciles high-performance and fault-tolerance, while being applicable to most transactional standards and products. This protocol was the final result of a series of contributions that rely on a novel paradigm for distributed transaction commit proposed in the context of this research.

7.1 Research Assessment

Divided into three parts, this thesis has led to six major contributions. The first part tackled performance issues, and introduced the *Dictatorial Atomic Commitment* problem, defined *On-line Serializability* and *On-line Commit-Resiliency*, and proposed a highly efficient commit protocol, named *Coordinator Logical Log* (CLL). The second part extended the previous results to cover fault-tolerance issues, and proposed two non-blocking extensions to CLL, which provide liveness guarantees under the two extremes of a spectrum of possible system models, namely synchronous and asynchronous systems. The third and final part addressed practical issues by describing a way by which the asynchronous non-blocking CLL variation can be integrated into existing transactional standards and products.

7.1.1 Performance Issues

Dictatorial Atomic Commitment. We have discussed some serious drawbacks of the traditional Two-Phase Commit (2PC) approach to the distributed commit problem, and argued that although it ensures transaction atomicity, 2PC introduces a substantial delay in the system, leading to a significant increase in transaction execution times. To meet the strong efficiency requirements of today's advanced and critical applications, and through a careful look into the characteristics of real-world transactional systems, we have identified the conditions under which a One-Phase Commit (1PC) approach can be used. Our research led us to define the *Dictatorial Atomic Commitment* (DAC) problem, a novel paradigm for distributed transaction termination, which overcomes the need for 2PC in most practical situations. Based on the pragmatic observation that, in most real settings, participants' votes can turn out to be more than necessary, the Dictatorial Atomic Commitment problem resulted from removing *veto rights* from the traditional Atomic Commitment problem.

In addition to defining Dictatorial Atomic Commitment, we have also proposed a simple algorithm that solves it based on a 1PC approach. This algorithm corresponds exactly to a 2PC without the voting phase, which explains why 1PC is much more efficient than 2PC. A crucial feature of our algorithm is that it constitutes the basic building block around which all existing 1PC variations are designed, thus promoting modularity.

On-line Serializability & On-line Commit-Resiliency. To characterize transactional systems that are compatible with dictatorial transaction processing, we have studied the impact of dictatorship on concurrency control and recovery protocols employed by the participants in a transaction. In particular, we have defined three necessary and sufficient conditions to ensure the correctness of transactional systems with no participant veto right: *on-line serializability*, *cascadelessness*, and *on-line commit-resiliency*. These conditions are strictly stronger than the usual correctness metrics for transactional systems, namely *serializability*, *recoverability* and *resiliency*, respectively. We have also addressed practical considerations related to those conditions, and have shown that, whereas *on-line serializability* and *cascadelessness* are realistic in most real settings, *on-line commit-resiliency* turned out to be very expensive in practice.

Coordinator Logical Log. To overcome the high cost imposed by *on-line commit-resiliency*, we have considered a "non-classical" atomic commitment scheme that allows participants to delegate part of their transactional responsibilities to the coordinator of the commit protocol. Through a deep analysis of existing 1PC variations that follow this scheme, we have pointed out their practical limitations when it comes to meeting autonomy requirements of today's distributed environments. In order to combine the high-efficiency of 1PC with practical utility, we have proposed a new 1PC variation, called *Coordinator Logical Log* (*CLL*), which preserves site autonomy based on a logical logging recovery mechanism. The advantages of CLL are not only performance issues. By eliminating participants' votes, and maintaining logical operations instead of physical log records at the coordinator site, CLL seems to be the sole protocol that can cope with all existing transactional systems, be they or not 2PC compliant.

7.1.2 Fault-tolerance Issues

Non-blocking Coordinator Logical Log. Although more efficient than the 2PC approach, 1PC increases the probability of blocking of transaction participants in case of failures, making the window of vulnerability to blocking last all along the transaction execution. While this might be acceptable in some standard applications, there are mission critical applications for which high-performance and fault-tolerance are crucial requirements of equal importance. The above observation constituted our starting point

for investigating solutions to the *Non-Blocking Dictatorial Atomic Commitment* (NB-DAC) problem. The first result of this study has been a variation of our CLL protocol, named *Non-Blocking CLL* (NB-CLL), which achieves non-blocking in the context of synchronous systems based on a *Uniform Timed Reliable Broadcast* (UTRB) primitive, and assuming reliable communication and reliable failure detection.

Asynchronous Non-blocking Coordinator Logical Log. Given that synchrony assumptions and reliable failure detectors are not always realistic in practice, we have extended our work on fault-tolerance and proposed a variation of CLL, called *Asynchronous Non-Blocking CLL* (ANB-CLL), which guarantees non-blocking assuming an asynchronous system with reliable communication and unreliable failure detectors. A crucial feature of ANB-CLL is that it achieves non-blocking based on a *crash-recovery* failure model. To the best of our knowledge, it is the first time that fault-tolerant solutions to the distributed commit problem have been designed in the context of asynchronous systems in which processes may crash and later recover. Furthermore, ANB-CLL blends the advantages of CLL in terms of efficiency and autonomy requirements with fault-tolerance, making it the best candidate for distributed transaction commit in the context of today's systems and applications.

7.1.3 Prototype Design & Implementation

Through a prototype design and implementation, we have shown how our ANB-CLL protocol can be integrated into well-known transactional standards. This prototype has served as a proof of concept, which shows the validity of our theoretical study, and the compliance of our protocol with current transactional standards and products.

Following a "compositional methodology" of protocol integration, the (blocking) CLL protocol has been first embedded into an OTS/DTP environment. We have then designed a non-blocking extension to the CLL prototype as a separate construct that can be added on top of it. At the time of writing of this thesis, the non-blocking extension has not yet been implemented and integrated into the prototype, and thus remains at its design stage.

7.2 Future Directions and Open Issues

In addition to the contributions presented in the previous section, several extensions to our work need to be explored, allowing plenty of scope for interesting research. In the following, we describe some future directions and open questions.

Towards a Higher Resiliency During Recovery. The Coordinator Logical Logging recovery mechanism associated with the different CLL variations preserves site autonomy, and overcomes the high cost introduced by *on-line commit-resiliency* at the expense, however, of a coordinator-dependent recovery protocol. An important future work would be to explore new coordination schemes that enable to increase the resilience of the recovery protocol by decreasing its dependency level. One intuitive way of achieving this would consist in replicating the coordinator's log at some other sites, which number depends on the desired resiliency rate. This actually lays the basis for further investigations related to the cost this might introduce in the system.

Deferred Consistency Constraints. One consequence of removing veto rights from transaction participants is that integrity constraints are checked after each update operation, and thus deferred integrity validation is excluded. An open question is then whether it is possible to circumvent this assumption so as to widen the applicability field of dictatorial transaction processing. This would probably consist in exploring intermediate schemes between *veto rights for all* and *no veto right at all*.

ANB-CLL for Mobile and Disconnected Computing. Mobile and disconnected computing is clearly one of the most challenging areas for future distributed environments. The growing number of applications using mobile and disconnected facilities, supported by the emerging world of lightweight intelligent devices, raises new issues in terms of transaction management and introduces new requirements that the traditional transaction processing paradigm cannot meet. For instance, a traditional (i.e., 2PC-like) commit protocol leads to the abort of a transaction after it has been successfully processed if any of its participants disconnects during the voting phase. This situation is rather intolerable in a mobile environment where (accidental or voluntary) disconnections are very frequent. Furthermore, by forcing participants in a transaction to

externalize local prepared states, traditional protocols consume valuable system resources on data servers hosted by lightweight devices.

In this context, our ANB-CLL protocol seems to cope effectively with these issues, and it would be very interesting to study its adaptation to mobile and disconnected computing environments. Indeed, by eliminating participants' votes and local prepared states, ANB-CLL provides a very suitable way of dealing with disconnections, and allows saving critical resources on lightweight servers. Although not yet totally conclusive, a preliminary study showed the appropriateness of our protocol in bringing answers to these issues through three typical mobile/disconnected computing applications [BPA00], but this still needs further investigation.

Bibliography

[ABG98] M. Abdallah, C. Bobineau, R. Guerraoui, P. Pucheral, "Specification of the Transaction Service", Deliverable n°R13, Esprit Project OpenDREAMS-II n°25262, July 1998.

[AbP97] M. Abdallah, P. Pucheral, "Validation Atomique: état de l'art et perspectives". *Ingénierie des Systèmes d'Information (ISI)*, 5(6), December 1997.

[AbP98a] M. Abdallah, P. Pucheral, "A Non-Blocking Single-Phase Commit Protocol for Rigorous Participants". *Networking and Information Systems Journal (NIS),* 1(2-3), April 1998. A preliminary version of this paper appeared in *13èmes journées Bases de Données Avancées (BDA)*, September 1997.

[AbP98b] M. Abdallah, P. Pucheral, "A Single-Phase Non-Blocking Commit Protocol". *Proc. of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, August 1998.

[AbP99] M. Abdallah, P. Pucheral, "A Low-Cost Non-Blocking Atomic Commitment Protocol for Asynchronous Systems". *Proc. of the 11th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, November 1999.

[AGP98] M. Abdallah, R. Guerraoui, P. Pucheral, "One-Phase Commit: Does It Make Sense?". *Proc. of the 6th International Conference on Parallel and Distributed Systems (ICPADS)*, December 1998.

[AGP00] M. Abdallah, R. Guerraoui, P. Pucheral, "Dictatorial Transaction Processing: Atomic Commitment without Veto Right". Technical Report 2000/6, PRiSM Laboratory, University of Versailles, April 2000.

[AlC95] Y. Al-Houmaily, P.K. Chrysanthis, "Two-phase Commit in Gigabit-Networked Distributed Databases", *Proc. of the 8th International Conference on Parallel and Distributed Computing Systems (PDCS)*, September 1995.

[AlC96] Y. Al-Houmaily, P.K. Chrysanthis, "The Implicit-Yes Vote Commit Protocol with Delegation of Commitment", *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems (PDCS)*, September 1996.

[ACL97] Y. Al-Houmaily, P. Chrysanthis, S. Levitan, "An Argument in Favor of the Presumed Commit Protocol", *Proc. of the 13th IEEE International Conference on Data Engineering (ICDE)*, April 1997.

[AlS85] B. Alpern, F.B. Schneider, "Defining liveness", *Information Processing Letters*, 21(4), October 1985.

[BaT93] O. Babaoglu, S. Toueg, "Non-Blocking Atomic Commitment", *Distributed Systems*, ACM Press, 1993.

[BBP00] C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, "PicoDBMS: Scaling down Database Techniques for the Smartcard", *Proc. of the 26th International Conference on Very Large Data Bases (VLDB)*, September 2000.

[BCF97] J. Besancenot, M. Cart, J. Ferrié, R. Guerraoui, P. Pucheral, B. Traverson, *Les systèmes transactionnels: concepts, produits et normes*, Hermès, 1997.

[BGS92] Y. Breitbart, H. Garcia-Molina, A. Silbershatz, "Overview of Multidatabase Transaction Management". *VLDB Journal*, 1(2), October 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[BPA00] C. Bobineau, P. Pucheral, M. Abdallah, "A Unilateral Commit Protocol for Mobile and Disconnected Computing", *Proc. of the 13th International Conference on Parallel and Distributed Computing Systems (PDCS)*, August 2000.

[Cha93] T.D. Chandra, "Unreliable Failure Detectors for Asynchronous Distributed Systems", Ph.D. dissertation, Graduate School of Cornell University, May 1993.

[CHT96] T.D. Chandra, V. Hadzilacos, S. Toueg, "The Weakest Failure Detector for Solving Consensus", *Journal of the ACM, 43(4), July 1996.*

[ChT96] T.D. Chandra, S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, 43(2), March 1996.

[DwS83] C. Dwork, D. Skeen, "The Inherent Cost of Non-Blocking Commitment", *Proc. of the 2nd ACM Symposium on Principles of Distributed Computing (PODC)*, August 1983.

[Fis83] M. J. Fischer, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)", Technical Report 273, Department of Computer Science, Yale University, June 1983.

[Fel98] P. Felber, "The CORBA Object Group Service: A Service Approach to Object Groups in CORBA", Ph.D. dissertation, Computer Science Department, Swiss Federal Institute of Technology, 1998.

[FLP85] M. J. Fischer, N. Lynch, M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM*, 32(2), April 1985.

[GLS95] R. Guerraoui, M. Larrea, A. Schiper, "Non-Blocking Atomic Commitment with an Unreliable Failure Detector", *Proc. of the 14th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, September 1995.

[GLS96] R. Guerraoui, M. Larrea, A. Schiper, "Reducing the Cost for Non-Blocking in Atomic Commitment", *Proc. of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1996.

[Gra78] J. Gray, "Notes on Database Operating Systems", *Operating Systems: An Advanced Course*, LNCS 60, Springer Verlag, 1978.

[Gra90] J. Gray, "A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem", *Fault-Tolerant Distributed Computing*, LNCS 448, Springer Verlag, 1990.

[GrR93] J. Gray, A. Reuter, *Transaction processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[Gue95] R. Guerraoui, "Revisiting the relationship between non-blocking atomic commitment and consensus", *Proc. of the 9h International Workshop on Distributed Algorithms (WDAG)*, September 1995.

[Gue96] R. Guerraoui, "Distributed Transactions: Algorithms, Systems and Languages", Research Supervision Habilitation dissertation (HDR), University of Grenoble, 1996.

[GuS95] R. Guerraoui, A. Schiper, "The Decentralized Non-Blocking Atomic Commitment Protocol", *Proc. of the 7h IEEE International Symposium on Parallel and Distributed Processing (SPDP)*, October 1995.

[Had88] V. Hadzilacos, "A Theory of Reliability in Database Systems", *Journal of the ACM*, 35 (1), January 1988.

[Had90] V. Hadzilacos, "On the Relationship Between the Atomic Commitment and Consensus Problems", *Fault-Tolerant Distributed Computing*, LNCS 448, Springer Verlag, 1990.

[HaM90] J.Y. Halpern, Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment", *Journal of the ACM*, 37 (3), July 1990.

[HaT94] V. Hadzilacos, S. Toueg, “A Modular Approach to Fault-tolerant Broadcasts and Related Problems”, Technical Report TR 94-1425, Department of Computer Science, Cornell University, May 1994.

[ION97] IONA, *Orbix 2.3 Programmer’s Guide*, IONA Technologies PLC, 1997.

[ISO92a] International Standardization Organization, *Information Technology -- Open System Interconnection -- Distributed Transaction Processing -- Part 1/2/3: Model/ Service/Protocol*, ISO/IEC 10026-1/2/3, 1992.

[ISO92b] International Standardization Organization, *Information Processing Systems -- Database Language SQL*, ISO/IEC 9075, 1992.

[KeD94] I. Keidar, D. Dolev, “Increasing the Resilience of Atomic Commit at No Additional Cost”, Technical Report CS94-18, Institute of Computer Science, The Hebrew University of Jerusalem, October 1994.

[LaF82] L. Lamport, M. Fisher, “Byzantine generals and transaction commit protocols”, *Technical Report 62, SRI International*, April 1982.

[LaL93] B. Lampson, D. Lomet, “A New Presumed Commit Optimization for Two Phase Commit”, *Proc. of the 19^h International Conference on Very Large Data Bases (VLDB)*, August 1993.

[Lam77] L. Lamport, “Proving the correctness of multiprocess programs”, *IEEE Transactions on Software Engineering*, 3 (2), March 1977.

[LSG98] J. Liang, M. Saheb, F. Giudice, “Maao OTS version2”, Deliverable n°D2aa, ACTS Project ACTranS, 1998. Available at http://www.actrans.org/Publications.html.

[LSP82] L. Lamport, R. Shostak, M. Pease, “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems*, 4 (3), July 1982.

[MLO86] C. Mohan, B. Lindsay, R. Obermarck, “Transaction Management in the R* Distributed Database Management System”, *ACM Transactions on Database Systems*, 11 (4), December 1986.

[OMG00a] Object Management Group, *Object Transaction Service*, Document 00-06-28, Version 1.1, 2000.

[OMG00b] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Version 2.4, Document 00-10-01, 2000.

[Pri94] F. Primatesta, *TUXEDO: An Open Approach to OLTP*, Prentice Hall, 1994.

[She93] M. Sherman, "Architecture of the Encina Distributed Transaction Processing Family", *Proc. of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[ShL90] A. Sheth, J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases", *ACM computing surveys*, 22 (3), September 1990.

[Ske81] D. Skeen, "NonBlocking Commit Protocols", *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1981.

[Ske82] D. Skeen, "A Quorum-Based Commit Protocol", *Proc. of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1982.

[StC90] J. Stamos, F. Cristian, "A Low-Cost Atomic Commit Protocol", *Proc. of the 9th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1990.

[StC93] J. Stamos, F. Cristian, "Coordinator Log Transaction Execution Protocol", *Journal of Distributed and Parallel Databases*, 1 (4), October 1993.

[Sto79] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering,* 5 (3), May 1979.

[X/Op91] X/Open Company Ltd., *X/Open CAE Specification -- Distributed Transaction Processing: the XA Specification*, C193, 1991.

[X/Op93] X/Open Company Ltd., *X/Open Guide -- Distributed Transaction Processing: Reference Model*, Version 2, G307, 1993.

Personal Publications

Refereed Journal Papers

[AGP00] M. Abdallah, R. Guerraoui, P. Pucheral, "Dictatorial Transaction Processing: Atomic Commitment without Veto Right". Accepted for publication in *Distributed and Parallel Databases Journal (DAPD)*.

[AbP98a] M. Abdallah, P. Pucheral, "A Non-Blocking Single-Phase Commit Protocol for Rigorous Participants". *Networking and Information Systems Journal (NIS),* 1(2-3), April 1998.

[AbP97] M. Abdallah, P. Pucheral, "Validation Atomique: état de l'art et perspectives". *Ingénierie des Systèmes d'Information (ISI)*, 5(6), December 1997.

Refereed Conference Papers

[AbP99] M. Abdallah, P. Pucheral, "A Low-Cost Non-Blocking Atomic Commitment Protocol for Asynchronous Systems". *Proc. of the 11th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, November 1999.

[AbP98b] M. Abdallah, P. Pucheral, "A Single-Phase Non-Blocking Commit Protocol". *Proc. of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, August 1998.

[AGP98] M. Abdallah, R. Guerraoui, P. Pucheral, "One-Phase Commit: Does It Make Sense?". *Proc. of the 6th International Conference on Parallel and Distributed Systems (ICPADS)*, December 1998.

[AbP98] M. Abdallah, P. Pucheral, "A Non-Blocking Single-Phase Commit Protocol for Rigorous Participants". *13èmes journées Bases de Données Avancées (BDA)*, September 1997. This publication is a preliminary version of [AbP98a].

[BPA00] C. Bobineau, P. Pucheral, M. Abdallah, "A Unilateral Commit Protocol for Mobile and Disconnected Computing", *Proc. of the 13th International Conference on Parallel and Distributed Computing Systems (PDCS)*, August 2000.

Reports

[ABG98] M. Abdallah, C. Bobineau, R. Guerraoui, P. Pucheral, "Specification of the Transaction Service", Deliverable n°R13, Esprit Project OpenDREAMS-II n°25262, July 1998.