How Fast can a Distributed Transaction Commit?

Rachid Guerraoui
École Polytechnique Fédérale de Lausanne
Station 14, CH-1015
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Jingjing Wang
École Polytechnique Fédérale de Lausanne
Station 14, CH-1015
Lausanne, Switzerland
jingjing.wang@epfl.ch

ABSTRACT

The atomic commit problem lies at the heart of distributed database systems. The problem consists for a set of processes (database nodes) to agree on whether to commit or abort a transaction (*agreement* property). The commit decision can only be taken if all processes are initially willing to commit the transaction, and this decision must be taken if all processes are willing to commit *and* there is no failure (*validity* property). An atomic commit protocol is said to be *non-blocking* if every correct process (a database node that does not fail) eventually reaches a decision (commit or abort) even if there are failures elsewhere in the distributed database system (*termination* property).

Surprisingly, despite the importance of the atomic commit problem, little is known about its complexity. In this paper, we present, for the first time, a systematic study on the time and message complexity of the problem. We measure complexity in the executions that are considered the most frequent in practice, i.e., failure-free, with all processes willing to commit. In other words, we measure *how fast a transaction can commit.* Through our systematic study, we close many open questions like the complexity of synchronous non-blocking atomic commit. We also present optimal protocols which may be of independent interest. In particular, we present an effective protocol which solves what we call *indulgent atomic commit* that tolerates practical distributed database systems which are synchronous "most of the time".

1. INTRODUCTION

The use of transactions to ensure the consistency of distributed databases systems despite concurrency and failures dates back to the 70's [1, 2, 3], and is still prominent today. Many modern distributed information systems are transactional, including HP's Sinfonia [4], Yahoo's PNUTS [5], Google's Percolator [6] and Spanner [7], Clock-SI [8] and Yesquel [9].

At the heart of those distributed transaction processing systems lies the fundamental *atomic commit* problem [2]. To illustrate the nature of the problem, consider a distributed database system that ensures the serializability of transactions by tracking their concurrency conflicts across datacenters (nodes) as in Helios [10]. In short, each datacenter $\mathcal{D}$ votes to abort every transaction tx that causes a conflict at $\mathcal{D}$. Transaction tx is committed if no datacenter detects any conflict involving tx. To orchestrate the termination of tx, coordination is necessary among datacenters: all have to agree on whether to commit or abort tx, despite failures, and tx cannot be committed if at least one datacenter votes to abort. This coordination is called a distributed commit protocol and its complexity impacts the performance of the entire distributed database system [10].

1.1 Problem statement

More specifically, the atomic commit problem consists for a set of nodes of the distributed database system (we simply call them *processes*) to decide whether to *abort* or *commit* a transaction. The decision is based on the *vote* of each process about the local faith of the transaction. A process votes "no" if the transaction did not execute correctly at that process (due to a full disk, a concurrency control problem, etc.). A process votes "yes" (willingness to commit) if the transaction did execute correctly at that process. The processes (a) commit the transaction *only* if all vote to commit, and (b) *have to* commit the transaction if all vote to commit and there is no failure. This property is usually called *validity* [11, 12, 13, 14, 15]. All processes need to agree on the same decision. This property is called *agreement* [11, 12, 13, 14, 15]. If one additionally stipulates that correct processes (those that do not crash) need to eventually decide (commit or abort) despite failures (e.g., crashes of other processes), then this property is called *termination* [14, 15], and the resulting problem, where processes need to ensure validity, agreement as well as termination, is called *non-blocking atomic commit* (NBAC) [16]. NBAC has been investigated since the 70's by the database and distributed system communities [16, 17, 18, 11, 19, 12, 20, 21].

In this paper, we present a systematic study of the time and message complexity of the atomic commit problem and study the exact tradeoff between robustness and *best-case* complexity (in the sense of Lamport [22]), i.e., the complexity of any failure-free execution where all processes vote to commit. Such executions, called *nice* executions in this paper, are arguably the most frequent in practice and are those for which protocols are usually optimized.

Not surprisingly, this complexity depends on *robustness*, i.e., on which property (validity, agreement, termination) is required in which executions (including less likely executions with failures). The most robust form of atomic commit protocol is, roughly speaking, the one that tolerates both *crash failures* (i.e., some process crashes) and *network failures* (e.g., a network partition occurs and later recovers), i.e., all executions with such failures have to solve NBAC. However, by the impossibility result of consensus [23, 24], this most robust form has infinite complexity. On the contrary, the least robust form of atomic commit, of which only *failure-free* executions are required to solve NBAC, is clearly easy to solve in finite complexity. Although there is obviously a tradeoff between robustness and complexity, the exact tradeoff was not clear. Furthermore, between the least and most robust forms of atomic commit, the situation is more complicated and the complexity results harder to obtain.

We exhaustively study complexity in the cases between two extremes, assuming certain robustness of an atomic commit protocol. More precisely, we determine the optimal number of message delays/messages in nice executions of a protocol π assuming that, in π, (1) every *crash-failure* execution satisfies X and (2) every *network-failure* execution satisfies Y, where X and Y are subsets of these three properties: agreement, validity, and termination. With two kinds of failure-prone executions (crash-failure and network-failure) and three properties, we end up with $(2^3)^2 = 64$ possibilities, as shown in Table 1. Since a property satisfied in every network-failure execution is also satisfied in every crash-failure execution, the 64 possibilities reduce to 27 different cases, the non-empty cells in Table 1.

1.2 Previous results

Many distributed database systems (Sinfonia [4], Percolator [6], Spanner [7], Clock-SI [8] and Yesquel [9], for instance) guarantee validity and agreement in crash-failure executions through a two-phase commit (2PC) protocol [2]. 2PC induces two communication rounds among processes. Although efficient, 2PC does not solve NBAC in crash-failure executions since it does not guarantee termination. However, NBAC can actually be solved in crash-failure executions (by a three-phase commit protocol [16], which has only finite complexity).

Except for some results on the number of messages necessary for synchronous NBAC protocols (which solve NBAC in every crash-failure execution) [17, 25, 26], the fundamental question of the complexity of synchronous NBAC has actually been open for more than three decades [16, 17]. In fact, only the lower bound of $2n-2$ messages in the face of $n-1$ crashes [17] was known until the present paper. Although important, little was known on the complexity of atomic commit (e.g., when network failures are also considered) or its tradeoff with robustness, which we address in this paper.

1.3 Our results

Table 1 summarizes our results for the 27 atomic commit problems considered. Besides the tradeoff between complexity and robustness (which properties are required in which execution), we also highlight a tradeoff between time and message complexity. We prove that in 18 out of 27 problem variants, the optimal number of message delays and the optimal number of messages cannot be achieved at the same time.

Among the 27 variants, the most robust one, which we call *indulgent atomic commit*, is particularly appealing.[1] Indulgent atomic commit captures the best robustness of a distributed commit protocol, i.e., despite failures, agreement, validity and termination are still satisfied. We propose a protocol, which we denote by INBAC, that matches the lower bound of two message delays of indulgent atomic commit. Moreover, we prove that INBAC is optimal in the number of messages among all delay-optimal indulgent atomic commit protocols. Thus, in practical distributed database systems that are synchronous "most of the time" [34][2], and where practitioners consider violations of timeouts (e.g., due to network failures), if rare, to be acceptable, INBAC tolerates such violations and is also optimal in complexity for the arguably most frequent executions. Comparing our INBAC protocol with the popular 2PC protocol, we show, interestingly, that (1) INBAC has the same best-case message delay as 2PC if all processes start spontaneously, and (2) in the special case where at most one process can crash (among n processes), INBAC and 2PC use $2n$ and $2n-2$ messages respectively. In this sense, INBAC may be of independent interest, as a more robust yet efficient alternative to 2PC for implementing distributed transactions.

At the same time, we close the question of the complexity of synchronous NBAC (which is one among the 27 cases we consider). We show, for the first time, that for synchronous NBAC, one message delay is optimal. We also generalize Dwork and Skeen's lower bound of $2n-2$ messages [17] to $n-1+f$ messages in the face of f crashes and propose a matching message-optimal synchronous NBAC protocol.

1.4 Techniques

We denote a cell in Table 1 by a property pair (X, Y). (X, Y) is less robust than another pair (U, V) if $X \subseteq U$ and $Y \subseteq V$. Then our proof goes through two main steps. First, we group the pairs (X, Y) that give the same number of message delays/messages in Table 1 and prove the lower bound for the least robust pair in each group. To design matching protocols, by symmetry, we look for "the most robust pair" in each group. However, as shown in Table 1, in some groups, there is no "most robust pair". Thus, our second step is to choose, in each group, the pairs that are locally maximal in robustness and present a protocol that matches the lower bound for each local maximum.

Three techniques are key to our results.

1. To prove our lower bounds, we introduce and leverage the notion of "process reachability", the arrival of a message m at process Q that makes Q

[1] We define indulgent atomic commit in the same vein as indulgent consensus [27, 28] protocols like Paxos [29], CHT [15] and others [30, 31, 32, 33].

[2] It was experimentally shown, e.g., in [34], that the latency of a communication round is below some seconds (most of the time) if the link does not lose too many messages.

Table 1: Complexity of Atomic Commit. NF = network-failure executions; CF = crash-failure executions; A = agreement; V = validity; T = termination. Fraction d/m in a cell (X, Y) means that the tight lower bounds are d message delays, m messages respectively if (1) every failure-free execution solves NBAC, (2) every crash-failure execution satisfies a set X of properties and (3) every network-failure execution satisfies a set Y of properties. For every empty cell (X, Y), there exists a non-empty cell (Z, Y) such that $X \cup Y = Z$.

CF NF	∅	A	V	T	AV	AT	VT	AVT
∅	$1/0$	$1/0$	$1/n-1+f$	$1/0$	$1/n-1+f$	$1/0$	$1/n-1+f$	$1/n-1+f$
A		$1/0$			$1/n-1+f$	$1/0$		$2/2n-2+f$
V			$1/2n-2$		$1/2n-2$		$1/2n-2$	$1/2n-2$
T				$1/0$		$1/0$	$1/n-1+f$	$1/n-1+f$
AV					$1/2n-2$			$2/2n-2+f$
AT						$1/0$		$2/2n-2+f$
VT							$1/2n-2$	$1/2n-2$
AVT								$2/2n-2+f$

know process P's vote, which is necessary in the context of a network-failure execution. (Dwork and Skeen [17] used "process coloring" in proving lower bounds for synchronous NBAC. Compared with our notion, theirs does not distinguish the arrival from the departure of a message, since they solely focus on crash-failure executions, featuring bounded message delays.)

2. To design our optimal protocols, we introduce and leverage "implicit" votes for the willingness to commit. For example, to achieve 0-message protocols, instead of receiving a message telling process P process Q's vote, P may know that Q votes 1 by *not* receiving a certain message. We support an optimal nice execution by a complex failure-free execution that aborts.

3. Another technique we use is "helping". To reach the smallest number of messages or message delays in any nice execution, if some failure occurs, then processes must ask for help. To enable helping, backing up votes at other processes is necessary while sometimes a message of acknowledgement (that confirms the success of the backup) is also necessary. Both are key ideas behind INBAC.

The rest of the paper is organized as follows. Section 2 presents the distributed database models we consider and defines the non-blocking atomic commit problem. Section 3 establishes our lower bounds. Section 4 describes atomic commit protocols that meet the lower bounds. Section 5 presents indulgent atomic commit, an overview of our protocol INBAC and a proof of its optimality. Section 6 discusses related work. For space limitation, we defer the details of our protocols, as well as some of our proofs, to the appendix.

2. MODELS AND DEFINITIONS

2.1 Processes and channels

We consider a set Ω of n processes $P_1, P_2, \ldots, P_n$ (sometimes also denoted by O, P, Q, R). Here processes represent database nodes. Processes communicate by exchanging messages, through the network.

We assume that no process deviates from its specification and at most $f, 1 \leq f \leq n-1$ processes can *crash.* After a process crashes, it does not send any message. If a process does not crash, it is said to be *correct.*

Communication channels do not modify, inject, duplicate or lose messages. Every message sent is eventually received.

2.2 Failures and executions

We assume *synchronous computation*: there is a known upper bound on the time to execute a local step, which includes the delivery of a message by a process, its local processing by that process, as well as the sending of a message as a consequence of that processing.

Communication is said to be *synchronous* if there is a known upper bound on message transmission delays. Communication is said to be *eventually synchronous* if the delay on message transmission might be unbounded but only until some, possibly unknown, *global stabilization time* (after which there is a known upper bound on delays).[3] We accordingly consider two kinds of system models (or simply systems): a synchronous system [17] and an eventually synchronous system [35], based on their respective assumptions on communication.

An execution of a synchronous system is either *failure-free* or has *crash failures*: either all processes are correct, or some process crashes, while all message transmission delays are smaller than some known upper bound which we denote by U. If, in some execution, some message transmission delay is greater than U, then the system is no longer synchronous: we say that a *network failure* occurs. An execution of an eventually synchronous system can be failure-free, has crash failures, or network failures. We call a *failure-free* execution an execution where no failure occurs, a *crash-failure* execution one execution of a synchronous system (where only crash failures are possible) and a *network-failure*

[3] Recall that, in an *asynchronous* system, without any communication bound, agreement problems like consensus and NBAC are impossible [24].

execution one execution of an eventually synchronous system (where network failures are also possible). We accordingly call a synchronous system and an eventually synchronous system, a crash-failure system and a network-failure system, respectively.

2.3 Non-blocking atomic commit

We consider the problem of non-blocking atomic commit (NBAC) in the classical sense of Skeen [16], which was later refined in [14, 15].

Definition 1 (NBAC [14, 15, 16]). A protocol π is an atomic commit protocol if π is defined by two events:

- *Propose*: $P_i, i = 1, 2, \dots, n$ proposes value $v = 1$ (vote "yes") or $v = 0$ (vote "no").
- *Decide*: $P_i, i = 1, 2, \dots, n$ outputs the decided value.

An execution of π solves NBAC if it satisfies the following three properties:[4]

- *Validity*: A process decides 0 only if some process proposes 0 or a failure occurs. A process decides 1 only if no process proposes 0.
- *Termination*: Every correct process eventually decides.
- *Agreement*: No two processes decide differently.

Given a system $\mathcal{S}$ (crash-failure or network-failure), π solves NBAC in $\mathcal{S}$ if every execution of π in $\mathcal{S}$ solves NBAC.

(Later in the paper, in Section 5, we will introduce our new variant of the problem: *indulgent atomic commit.*)

A comparison with previous definitions from the literature is now in order. A synchronous NBAC protocol [16, 17] is a protocol which solves NBAC in a crash-failure system (and thus the complexity is covered by our study). In previous impossibility results [14, 36, 37, 27, 28], the definition of validity depended on which failure may occur. *(Strong) validity* was considered in the only case of crash failures, whereas a weak form of validity, *weak validity*,[5] was distinguished if a failure could be a network failure. Definition 1 unifies validity and weak validity for presentation clarity and consistency with previous impossibility results.

2.4 Complexity measures

We define a *nice execution* of an atomic commit protocol as a failure-free execution in which every process proposes 1. We study in this paper *best-case complexity*, i.e., the complexity over nice executions (which are arguably the most frequent in practice). We consider two complexity measures: the number of messages and the number of message delays. Here (as in Lamport [22, 38]), for any message m, one message delay is a period of time between two events: the sending of m and the reception of m [22, 38]. Thus if local computation is instantaneous (negligible), and every message is received exactly one unit of time after it was sent, then the number of message delays of an execution is the number of units of time of that execution [22].

3. LOWER BOUNDS

In this section, we establish lower bounds on the number of message delays, and then lower bounds on the number of messages.

3.1 Message delays

As shown in Table 1, there are two possibilities for the lower bound on the number of message delays: 1 and 2. There are four non-empty cells in Table 1 of which the lower bound is 2: (AVT, A), (AVT, AV), (AVT, AT), and (AVT, AVT). Among them, (AVT, A) is the least robust. The rest of the non-empty cells have 1 as the lower bound, among which ($\emptyset$, $\emptyset$) is the least robust. Thus we need only to prove lower bounds for two cells: ($\emptyset$, $\emptyset$), (AVT, A) respectively, as summarized in Theorem 1.

Theorem 1 (Lower bound on message delays). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be any two subsets of $\mathcal{P} = \{$agreement, validity, termination$\}$. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathcal{P}_1$ in every crash-failure execution and (c) satisfies $\mathcal{P}_2$ in every network-failure execution. Let d be the smallest number of message delays among all nice executions of π. If for π, $\mathcal{P}_1 = \mathcal{P}_2 = \emptyset$, then $d \geq 1$. If for π, $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_2 = \{$agreement$\}$, then $d \geq 2$.*

The proof of the first part of Theorem 1 is immediate: to satisfy validity in every failure-free execution, no process can decide immediately; i.e., the process has to wait for at least one message delay to know other processes' votes.

The proof of the second part is less obvious, and goes through an intermediary lemma. This lemma makes use of the notion of "process reachability", which we introduce here and use in all our lower bound proofs.

Definition 2 (Reaching a process). If a protocol instructs a process *src* to send a message m to another process *dest*, then we say that *src* is the *source* of m and *dest*, the *destination* of m. Let E be any execution. In E, if *src* sends m at time t, then we may interchangeably say that m *leaves* from *src* (for *dest*) at t; if at time t, *dest* receives m, then we may interchangeably say that m *arrives* at *dest* at t.

Let $\underline{m} = \{m_1, m_2, \dots, m_l\}$ be a sequence of messages such that (a) the source of m_1 is P, (b) the destination of m_l is $Q, Q \neq P$, (c) the source src_i of m_i is the destination of m_{i-1} for $i = 2, 3, \dots, l$, and (d) m_i leaves from src_i later than or at the time at which m_{i-1} arrives at src_i for $i = 2, 3, \dots, l$. If m exists for two processes

[4] An execution of an atomic commit protocol also satisfies a property called *integrity*, i.e., no process decides twice in any execution. This is immediate to satisfy in our context so we omit it for presentation simplicity.

[5] Weak validity allows processes to abort a transaction (decide 0) even if none of them crashes and all of them vote to commit (propose 1), as long as there is a network failure.

P, Q and $l \geq 1$ in E, then we say that P *reaches* Q in E.

If m_l arrives at Q at time t or earlier and $\underline{m}$ is the earliest sequence of messages for P (according to t) to reach Q in E, then we say that P *has reached* Q at time t in E.

If a process P reaches another process Q, then it is possible that, by a sequence of messages, P *backs up* P's vote at Q. The intuition of the lower bound in question, captured by Lemma 1 below, is then for P's vote, at least f backups are necessary.

Lemma 1 (Backups). *Let π be any protocol that solves NBAC in every crash-failure execution and ensures agreement in every network-failure execution. Let E be any nice execution of π. Let P decide at time t_1 in E. Among the messages whose destination is P, let $\mathcal{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathcal{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m \in \mathcal{M}} t_m$.*

Then at t_2, P has reached at least f processes.

Proof. By contradiction. Suppose that at t_2, P has reached at most $f - 1$ processes. Denote by Φ the set of those processes together with P. For any process $Q \in \Phi \backslash \{P\}$, denote by τ_Q the time at which P reaches Q. We build a crash-failure execution E_0 based on E. In E_0, P crashes at time 0 (before sending any message). For Q, E_0 is the same as E until Q crashes at τ_Q (before sending any message that is expected to send upon the message received by Q at τ_Q). In addition, we assume that every message sent after t_2 arrives later than t_1. Moreover, in E_0, P proposes 0, every process other than P proposes 1 and no process in $\Omega \backslash \Phi$ crashes. As $|\Phi| \leq f$ and $t_1 - t_2 \leq U$, E_0 is a legitimate crash-failure execution of π.

Since π solves NBAC in every crash-failure execution, any remaining process $R \in \Omega \backslash \Phi$ decides 0 in E_0. W.l.o.g., let R be the earliest process that decides. Denote by t_3 the time at which R decides. Then we build a network-failure execution E_{async} based on E and E_0. In E_{async}, every process proposes 1 and no process crashes. We construct E_{async} such that E_{async} starts as E and:

- Every message from P to a process in $\Omega \backslash \Phi$ arrives later than $\max(t_1, t_3)$;
- Every message from Q to a process in $\Omega \backslash \Phi$ sent after or at time τ_Q arrives later than $\max(t_1, t_3)$;
- Every message sent after t_2 arrives later than t_1.

To any process in $\Omega \backslash \Phi$, before and at t_2, E and E_{async} are indistinguishable. Since every message sent after t_2 arrives later than t_1, therefore E_{async} and E are indistinguishable for P before and at t_1. As a result, P decides 1 at t_1. For R, E_{async} is the same as E_0 before and at t_3. As a result, R decides 0 at t_3.

Clearly, E_{async} is a network-failure execution of π that does not satisfy *agreement*. A contradiction. □

Using Lemma 1, we now count the necessary number of message delays.

Proof. (Proof of the second part of Theorem 1.) Let t_2 be defined as in Lemma 1 for the earliest process P that decides in any nice execution. Then for $f \geq 1$, by Lemma 1, at t_2, at least one message from P must have arrived while another message just leaves from its source for P. This, in total, gives at least two message delays before any process decides. □

3.2 Messages

As shown in Table 1, there are four possibilities for the lower bound on the number of messages: 0, $n-1+f$, $2n-2$ and $2n-2+f$. We group the cells in Table 1 with the same value, and then prove the lower bound for the least robust atomic commit in each group. Thus we need only to prove lower bounds for four cells in Table 1: (∅, ∅), (V, ∅), (V, V), and (AVT, A) respectively, as summarized in Theorem 2.

Theorem 2 (Lower bounds on messages). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be any two subsets of $\mathcal{P} = \{agreement, validity, termination\}$. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathcal{P}_1$ in every crash-failure execution and (c) satisfies $\mathcal{P}_2$ in every network-failure execution. Let m be the smallest number of messages among all nice executions of π. If for π, $\mathcal{P}_1 = \mathcal{P}_2 = \emptyset$, then $m \geq 0$. If for π, $\mathcal{P}_1 = \{validity\}$ and $\mathcal{P}_2 = \emptyset$, then $m \geq n-1+f$. If for π, $\mathcal{P}_1 = \mathcal{P}_2 = \{validity\}$, then $m \geq 2n-2$. If for π, $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_2 = \{agreement\}$, then $m \geq 2n-2+f$.*

The proof of 0 message for the cell (∅, ∅) is trivial and omitted. In what follows, we count the number of necessary messages in the other three cases separately.

Lower bound of $n-1+f$ messages. We generalize here the lower bound of $2n-2$ messages for synchronous NBAC from Dwork and Skeen [17]. As in their proof, we first present a preliminary lemma, Lemma 2, which we phrase here in terms of "process reachability". As Lemma 2 is a (straightforward) generalization of the preliminary lemma in Dwork and Skeen's proof, the proof of Lemma 2 is omitted.[6]

Lemma 2 (Validity despite crashes). *Let π be any protocol that (a) solves NBAC in every failure-free execution and (b) ensures validity in every crash-failure execution. Then in any nice execution of π, every process reaches at least f processes.*

We then count the number of necessary messages. By Lemma 2, every process has to reach at least f processes in every nice execution, which gives at least $n-1+f$ messages to be exchanged. (Lemma 2 is similar to Lemma 1 except that the former claims the necessity

[6] We note, however, that the original preliminary lemma in [17] does not distinguish between the necessity of sending a message (before a certain point in time) and the necessity of receiving a message (before a certain point in time). It is thus only appropriate for a crash-failure execution (as after a message m is sent, it is predictable that m is received within some time period) and does not apply, as is, to the setting of a network-failure execution as we study in the rest of the paper. Hence the need to rephrase the preliminary lemma.

of f backups while the latter claims the necessity of f backups before a certain message.)

Lower bound of $2n-2$ messages. Before counting the number of necessary messages, we introduce a preliminary lemma.

Lemma 3 (Validity in every execution). *Let π be any protocol that that (a) solves NBAC in every failure-free execution and (b) ensures validity in every network-failure execution. Then in every nice execution of π, for any process Q, every other process P reaches Q before or when Q decides.*

Proof. By contradiction. Consider a nice execution E with two processes P and Q such that P has not reached Q when Q decides 1. Let Q decide at time t. Let Φ be the set of processes which P has reached before or at t. For every $R \in \Phi$, let τ_R be the time at which P reaches R. Now we build a network-failure execution E_{async} based on E. In E_{async}, P crashes before sending any message, P votes 0 and every other process votes 1. Clearly, for every process (except P), E_{async} starts as E; then for every $R \in \Phi$, we let every message from R sent at or after τ_R arrive later than t. Since in E, Q does not expect any message from R sent at or after τ_R and Q does not expect any message from P either, then Q does not distinguish E and E_{async} and thus decides 1 at t again in E_{async}, which violates validity. □

Now we count the number of necessary messages. Let R be the latest process that decides in a nice execution. By Lemma 3, before or when R decides, for any process Q, every other process $P \neq Q$ has reached Q. As a result, before or when R decides, at least $2n-2$ messages are exchanged, which gives the necessary number of messages.

We note that for atomic commit problems with $n-1+f$ messages and $2n-2$ messages as lower bounds, the lower bound on the number of message delays is 1. It is easy to show that the lower bound on the number of messages and that on the number of message delays cannot achieved at the same time: all those problems feature *validity* at least in every crash-failure execution and thus a 1-delay protocol must use at least $n(n-1)$ messages. This shows that for those problems (14 cases among totally 27 ones which we consider), there is a tradeoff between the number of messages and that of message delays.

Lower bound of $2n-2+f$ messages. Before counting the number of necessary messages, we again introduce a preliminary lemma.

Lemma 4 (Agreement in every execution). *Assume $f \geq 2$. Let π be any protocol that solves NBAC in every crash-failure execution and ensures agreement in every network-failure execution. Let E be any nice execution. Let P decide at time t_1 in E. Among the messages whose destination is P, let $\mathcal{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathcal{M}$, let t_m be the time at which m leaves from its source and $t_{2,P} = \max_{m\in\mathcal{M}} t_m$.*

Then at $t_{2,P}$ in E, every process has reached at least $f-1$ processes.

This is the same class of protocols as considered in Lemma 1. Lemma 4 is similar to Lemma 1 with the following difference: for every vote, at least $f-1$ backups are necessary (while Lemma 1 considers the backup of P's vote alone). The proof of Lemma 4 is also similar to that of Lemma 1, which is omitted for space limitation.

Then we count the number of necessary messages. Let $t_{2,P}$ be defined as in the statement of Lemma 4 for any process P in any execution. Let $t_2 = \min_{P\in\Omega} t_{2,P}$. Then at and after t_2, at least n messages have to leave their sources respectively. Since at t_2, every process has reached at least $f-1$ processes, then before or at t_2, at least $n-2+f$ messages have arrived at their destinations respectively. Therefore, at least $2n-2+f$ messages are exchanged in every nice execution.

Table 2: Delay-optimal Protocols. 1NBAC is a synchronous NBAC protocol. Each protocol achieves its lower bound in every nice execution.

Cell	AV, AV	AT, AT	AVT, VT	AVT, AVT
Prot.	avNBAC	0NBAC	1NBAC	INBAC

4. MATCHING PROTOCOLS

In this section, we prove the tightness of the lower bounds by presenting matching commit protocols. We do this for the number of message delays and the number of messages separately.

4.1 Delay-optimal protocols

Recall that in Table 1, there are two possibilities for the lower bound on the number of message delays: 1 and 2. Recall also that there are four cells in Table 1 of which the lower bound is 2: (AVT, A), (AVT, AV), (AVT, AT), and (AVT, AVT), among which the last one is the most robust. The rest of the non-empty cells correspond to a lower bound of 1 delay, among which (AV, AV), (AT, AT) and (AVT, VT) are three local maximum by the relation of robustness. Thus we need only to present delay-optimal protocols for four cells, as summarized in Table 2 as well as in Theorem 3.

Theorem 3 (Delay-optimal protocols). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be any two subsets of $\mathcal{P}$ = {agreement, validity, termination}. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathcal{P}_1$ in every crash-failure execution and (c) satisfies $\mathcal{P}_2$ in every network-failure execution. Let d be the smallest number of message delays among all nice executions of π. If $d = 1$, then it is possible that $\mathcal{P}_1 = \mathcal{P}_2$ = {agreement, validity}, or $\mathcal{P}_1 = \mathcal{P}_2$ = {agreement, termination}, or $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_2$ = {validity, termination}. If $d = 2$, then it is possible that $\mathcal{P}_1 = \mathcal{P}_2 = \mathcal{P}$.*

Among the protocols of Table 2, INBAC solves what we call *indulgent atomic commit*, which we will discuss in Section 5. 0NBAC is an optimal protocol also for the number of messages, which we will discuss with other message-optimal protocols. For space limitation, we only sketch the other two protocols here.

1NBAC. During a failure-free execution of 1NBAC, a process (a) sends its vote to every process, (b) collects all n votes, (c) sends the logical AND of all n votes to every process, and then (d) decides. Thus in every failure-free execution (as well as in every nice execution), every process decides the logical AND of all n votes within one message delay. It is easy to verify that every failure-free execution solves NBAC.

In other executions, every process starts by sending its vote to every (other) process, but then since failures may occur, a process P may collect fewer than n votes at the end of the first message delay. If so, P waits for the logical AND of all n votes sent (at the end of the first message delay) by another process. Denoted by [D, d] the message that contains the logical AND of all n votes. If P receives any [D, d] before or at the end of the second message delay, then P proposes d to consensus *uc*; otherwise, P proposes 0 to *uc*. Then P waits for a decision *dec* of *uc*. Here and later in this paper, we consider the consensus as defined in Definition 5 (which is deferred to Section 5).[7] By the *termination* property of consensus in a network-failure execution, *uc* eventually decides *dec*. Then P decides the same *dec* for 1NBAC. The full description of 1NBAC and its full proof of correctness are deferred to Appendix D.

avNBAC. As 1NBAC, avNBAC starts by having every process send its vote to every other process. Unlike 1NBAC, avNBAC does not require termination if a failure occurs; thus every process decides if and only if it collects all the votes at the end of the first message delay. The full description of avNBAC, which is similar to that of 1NBAC, is omitted.

4.2 Message-optimal protocols

As shown in Table 1, there are four lower bounds on the number of message delays: 0, $n-1+f$, $2n-2$, and $2n-2+f$. Similarly, we group the cells of which the lower bound takes the same value in Table 1, and find the most robust one or the local maximum in each group. Thus we need only to present message-optimal protocols for six cells, as summarized in Table 3 as well as in Theorem 4.

Table 3: Message-optimal Protocols. Name avNBAC is abused as the meaning is clear in the context. Protocol (n-1+f)NBAC is a synchronous NBAC protocol. Each protocol achieves its lower bound in every nice execution.

Cell	AT, AT	AV, A	AVT, T
Prot.	0NBAC	aNBAC	(n-1+f)NBAC
Cell	AV, AV	AVT, VT	AVT, AVT
Prot.	avNBAC	(2n-2)NBAC	(2n-2+f)NBAC

Theorem 4 (Message-optimal protocols). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be any two subsets of $\mathcal{P}$ = {agreement, validity, termination}. Let π be any protocol that (a) solves NBAC in every failure-free execution, (b) satisfies $\mathcal{P}_1$ in every crash-failure execution and (c) satisfies $\mathcal{P}_2$ in every network-failure execution. Let m be the smallest number of messages among all nice executions of π. If $m = 0$, then it is possible that $\mathcal{P}_1 = \mathcal{P}_2$ = {agreement, termination}. If $m = n-1+f$, then it is possible that $\mathcal{P}_1$ = {agreement, validity} and $\mathcal{P}_2$ = {agreement}, or $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_2$ = {termination}. If $m = 2n-2$, then it is possible that $\mathcal{P}_1 = \mathcal{P}_2$ = {agreement, validity}, or $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_2$ = {validity, termination}. If $m = 2n-2+f$, then it is possible that $\mathcal{P}_1 = \mathcal{P}_2 = \mathcal{P}$.*

For space limitation, we sketch only 0NBAC, (n-1+f)-NBAC, and (2n-2)NBAC, since avNBAC is similar to (2n-2)NBAC while aNBAC and (2n-2+f)NBAC are close to (n-1+f)NBAC. The full descriptions of the protocols and their full proofs of correctness are deferred to Appendix E.

0NBAC. At the beginning of any execution of this protocol, a process which votes 1 does not send any message, while a process which votes 0 sends [V, 0] to every (other) process. As a result, after one message delay, n processes are divided into three categories: (1) those who vote 0, (2) those who vote 1 and receive [V, 0], and (3) those who vote 1 but do not receive any message. The last category thus decides 1 at the end of the first message delay, while the other two propose a value to consensus *uc* later. The second category also sends [B, 0] to every other process. Those who receive [V, 0] or [B, 0] but have not decided yet acknowledge to the corresponding sender. Therefore, if a process in categories (1) and (2) receive $n-1$ acknowledgements, then it proposes 0 to *uc*, and 1 otherwise. Note that since the third category does not acknowledge, if the third category is not empty, then all processes would decide 1 in this case, satisfying agreement.

For best-case complexity, it is easy to see that in every nice execution, no message is ever sent, and furthermore, every process decides after one message delay. 0NBAC achieves the lower bound on the number of messages and that on the number of message delays at the same time. As a result, for the 4 cases (among 27 cases) covered by this protocol (using the robustness relation), no tradeoff is necessary.

(n-1+f)NBAC. During every nice execution of this protocol, the communication steps among processes is totally ordered. The totally-ordered sequence is: P_1, $P_2, \ldots, P_n$ and subsequently $P_1, P_2, \ldots, P_f$. Then (a) P_1 starts by sending P_1's vote to P_2; (b) each process in the sequence, upon the receipt of its predecessor's message, sends the collection of the votes so far to its successor except P_f which is at the end of the sequence; (c) (after the latest time at which P_f may receive its

[7] Consensus in this sense is sometimes called *uniform consensus* in the literature [39].

message) every process noops for $f+1$ message delays; and (d) during nooping, a process does not receive any message and thus decides 1. Then during every nice execution, $n-1+f$ messages are exchanged.

In other executions, if a process votes 0, then the process does not send any message to the successor when it first occurs in the sequence. If a process P does not receive its predecessor's message, then P does not send any message to its successor as well except that P is in the suffix $P_n, P_1, P_2, \ldots, P_f$. In the suffix, if P does not receive its predecessor's message or receives 0 from its predecessor, then P sends 0 to every other process. Subsequently, if any process receives a message of 0, then the process sends 0 again to every other process. At the end, if a process has ever received a message of 0, then it decides 0 (and 1 otherwise).

(2n-2)NBAC. During every nice execution, (a) every process sends its vote to P_n spontaneously, (b) then P_n sends the logical AND of all n votes to every process and (c) every process noops for $f+1$ message delays and decides 1. In every crash-failure execution, while nooping, if a process does not receive any message from P_n or receives a message of 0 from any process, then the process sends 0 to every process. In every execution, a process decides at the end of nooping. A process decides 1 only if it finds all n votes to be 1 from P_n and no message of 0 arrives during nooping. Nooping for $f+1$ message delays ensures that at least one process succeeds in notifying every correct process in any crash-failure execution, to ensure agreement.

5. INDULGENT ATOMIC COMMIT

In this section, we present our INBAC protocol. INBAC solves indulgent atomic commit as defined below. We believe this protocol to be of practical relevance for it is suited to practical distributed database systems which are synchronous "most of the time".

Definition 3 (Indulgent atomic commit)**.** A protocol π solves *indulgent atomic commit* if it satisfies the following:

- Every network-failure execution of π solves NBAC.

Indulgent atomic commit is the most robust atomic commit problem in Table 1. For this problem, we show that our INBAC protocol is optimal in the number of message delays, as well as in the number of messages given that optimal number of message delays. To give the intuition behind the optimal protocol, we first prove the lower bounds on the number of messages, and then sketch the optimal protocol. For space limitation, the full description of our INBAC protocol is deferred to Appendix A.

5.1 Lower bounds

We prove a lower bound on the number of messages exchanged given two message delays (which is optimal as shown in Theorem 1) during any nice execution actually for a less robust problem (than indulgent atomic commit), as stated in the following theorem.

Theorem 5 (Lower bound on messages given fewer than three message delays)**.** *Let π be any protocol that (a) solves NBAC in every crash-failure execution, and (b) satisfies agreement in every network-failure execution. Let E be any nice execution of π where every message is transmitted after an exact message delay U. W.l.o.g., E starts at time 0. If every process decides at or before $2U$ in E, then at least $2fn$ messages are exchanged in E.*

Note that for this less robust problem as well as indulgent atomic commit, $2n-2+f$ messages are optimal. Thus Theorem 5 also demonstrates the tradeoff between the number of messages and that of message delays for this less robust problem, indulgent atomic commit and other related problems (in total 4 cases out of 27 ones which we consider).

To prove the lower bound of $2fn$ messages, we look for two non-overlapping sets of messages Λ_1 and Λ_2 such that every message in Λ_1 precedes some message in Λ_2 in any nice execution. To describe the relation between those messages precisely, we again apply the notion of "process reachability" introduced in Definition 2 and complete the terminology.

Definition 4 (Reaching a process: complete terminology)**.** Let E be any execution. Let $\underline{m} = \{m_1, m_2, \ldots, m_l\}$ be a sequence of messages in E such that (a) the source of m_1 is P, (b) the destination of m_l is $Q, Q \neq P$, (c) the source src_i of m_i is the destination of m_{i-1} for $i = 2, 3, \ldots, l$, and (d) m_i leaves from src_i later than or at the time at which m_{i-1} arrives at src_i for $i = 2, 3, \ldots, l$.

Recall that (as defined in Definition 2) if m_l arrives at Q at time t or earlier and $\underline{m}$ is the earliest sequence of messages for P (according to t) to reach Q in E, then we say that P *has reached* Q at time t in E.

For any two processes P and Q, if there are two sequences of messages $\underline{m}^1 = m^1_1, m^1_2, \ldots, m^1_{l_1}$ and $\underline{m}^2 = m^2_1, m^2_2, \ldots, m^2_{l_2}$ such that (a) the source of m^1_1 and the destination of $m^2_{l_2}$ is P, (b) the source of m^2_1 and the destination of $m^1_{l_1}$ is Q, (c) m^2_1 leaves from Q later than or at the time at which $m^1_{l_1}$ arrives at Q, and (d) $m^2_{l_2}$ arrives at some time t or earlier, then we say that P reaches Q and *subsequently* Q reaches P before time t (including t).

More generally, given three processes P, Q and R, if there are two sequences of messages $\underline{m}^1 = m^1_1, m^1_2, \ldots, m^1_{l_1}$ and $\underline{m}^2 = m^2_1, m^2_2, \ldots, m^2_{l_2}$ such that (a) the source of m^1_1 is R, (b) the destination of $m^2_{l_2}$ is P, (c) the source of m^2_1 and the destination of $m^1_{l_1}$ is Q, (d) m^2_1 leaves from Q later than or at the time at which $m^1_{l_1}$ arrives at Q, and (e) $m^2_{l_2}$ arrives at some time t or earlier, then we say that R reaches Q and *subsequently* Q reaches P before time t (including t).[8]

[8] The time t mentioned in Definition 4 is only for convenience of our proof: the time is assumed to be an accurate global clock, but no process necessarily has access to the global clock.

Recall that if a process P reaches another process Q, then it is possible that by a sequence of messages, P backs up P's vote at Q. (Lemma 1 actually captures the intuition of backups.) Similarly, if P reaches Q and subsequently Q reaches P, then it is possible that by a sequence of messages, Q *acknowledges* the backup of P's vote at Q. Then Lemma 5 below essentially says that at least f processes must send acknowledgements that confirm the success of the backup, the intuition for our proof of lower bound.

Lemma 5 (Quick acknowledgements)**.** *Let π be any protocol that (a) solves NBAC in every crash-failure execution, and (b) satisfies agreement in every network-failure execution. Let E be any nice execution of π. Let P decide at some time t_1 in E. Let Θ be such set of processes that $\forall Q \in \Theta$, Q satisfies that before t_1 (including t_1) in E, P reaches Q, and subsequently Q reaches P. Among the messages whose destination is P, let $\mathcal{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathcal{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m\in\mathcal{M}} t_m$.*

If $t_2 \leq 2U$, then $|\Theta| \geq f$.

The sufficient condition in Lemma 5 is actually non-trivial: if P decides in *two or three message delays*, then f acknowledgements for one backup are necessary.[9]

Proof. (Proof sketch of Lemma 5. The full proof is deferred to Appendix C.1.) The number of acknowledgements is closely related to how quickly P can decide in a nice execution. In a network-failure system, P may be incorrectly detected as having crashed, be unaware of the incorrect detection and still decide quickly. Suppose that only s, for some $s < f$, acknowledgements are sufficient for decision. Then P's quick decision might put agreement at risk, since if all those s processes as well as P crash, no process will be able to recover P's quick decision.

However, if P decides slowly and P expects a message from some process R in order to decide, then some process Q^- might notice the crash detection of P (or Q). (We consider the worst scenario where $n \geq 3$ here. If $n \leq 2$, then the proof could be easier.) Then Q^- might report it to P via R, and as a result, P may notice the incorrect crash detection of itself and wait for others (instead of taking a decision). If $t_2 \leq 2U$, then there is no guarantee for Q^- to inform P of this incorrect detection in time since Q^- notices the detection at the earliest at U, Thus $t_2 \leq 2U$ is indeed a sufficient condition for P to be considered quick to require f acknowledgements. □

Given Lemma 5, it is easy to see that certain messages do follow an order in any nice execution and because of the inherent order, there exist two non-overlapping sets of messages in any nice execution of a 2-delay protocol. We now prove our Theorem 5, the lower bound on the number of messages.

Proof. (Proof of Theorem 5.) Consider any process P and let t_1 be the time at which P decides. Among the messages whose destination is P, let $\mathcal{M}$ be the set of messages that arrive at P before or at t_1. For each $m \in \mathcal{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m\in\mathcal{M}} t_m$. Then $t_2 = U$ and $t_1 = 2U$. By Lemma 1, at least f messages leave from P at time 0, and by Lemma 5, at least f messages arrive at P at time $2U$. This, in total, gives at least $2fn$ messages during any nice execution. □

5.2 Optimal protocol

We present here a protocol, which we denote INBAC, and which is delay-optimal as well as message-optimal given the optimal number of message delays.

We start by looking at what happens in nice executions of INBAC (which actually follows Lemma 1 and Lemma 5); then we explain in other executions, how INBAC uses an underlying consensus module to solve agreement. The module solves consensus in a network-failure system [23, 35, 12], which we recall in Definition 5. Many solutions to consensus have been devised, e.g., Paxos and its variants [29, 40] , but the correctness of INBAC or the best-case complexity of it does not rely on a particular algorithm. The modular approach (using consensus as a service) has been also taken in other distributed algorithms [21, 41].

The state transition of a process in both executions (nice or not) is illustrated in Figure 1. For space limitation, the detailed protocol, the proof of its correctness, and the proof of one necessary lemma for the design of the protocol are deferred to Appendix A, Appendix B, and Appendix C.2, respectively.

Definition 5 (Consensus [23])**.** A *consensus* protocol is defined by two events: *propose*, by which a process proposes a value $v = 0$ or 1, and *decide*, which outputs a decision to the process; furthermore, every execution satisfies the following properties: *termination*, *agreement* (similar to those properties of NBAC) and the following *validity* property:

- *Validity*: If a process decides v, then v was proposed by some process.

For simplicity, for time $2U$ or earlier in INBAC, every process sends a message or decides at multiples of U, i.e., at time 0, U or $2U$.

Overview of INBAC.
- Nice execution. Every nice execution E of INBAC starts by $P_1, P_2, \ldots, P_n$ sending their votes simultaneously. At time 0, every process P sends P's vote to f processes. We say that those f processes are P's *backup processes*. At time U, each of P's backup processes sends P's vote back to P as an acknowledgement. INBAC chooses the set $\mathcal{B}_P$ of P's backup processes as follows: for $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$, $\mathcal{B}_P = \{P_1, P_2, \ldots, P_f\}$; for $P \in \{P_1, P_2, \ldots, P_f\}$, $\mathcal{B}_P = \{P_1,$

[9] Yet the question whether f acknowledgements are necessary if a process decides after more than three message delays remains open. The reason why it may be possible to use fewer than f acknowledgements is also explained in the proof sketch of Lemma 5.

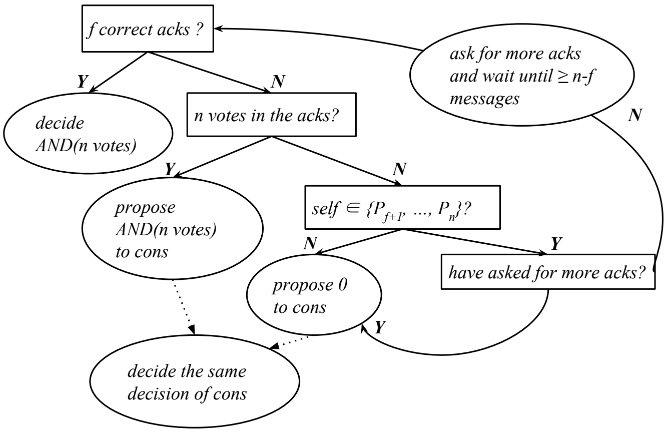
f correct acks ?
Y
N
decide AND(n votes)
n votes in the acks?
ask for more acks and wait until ≥ n-f messages
N
Y
N
propose AND(n votes) to cons
self ∈ {P_{f+1}, ..., P_n}?
N
Y
propose 0 to cons
have asked for more acks?
Y
decide the same decision of cons

Figure 1: State transition after $2U$

$P_2, \ldots, P_{f+1}\}\backslash\{P\}$. Clearly, a process may backup more than one vote. In fact, at time U, P's backup process sends to P a set $\mathcal{V}$ of the votes received as an acknowledgement of the successful backup of each vote in $\mathcal{V}$. (This is a necessary design, which we summarize later in Lemma 6). Thus at time $2U$, P decides if P receives f correct acknowledgements (from P's f backup processes where a correct acknowledgement from process $B \in \mathcal{B}_P$ includes Q's vote for all Q such that $B \in \mathcal{B}_Q$). Obviously, in a nice execution, or more generally, in an execution where messages arrive in time, at $2U$, P knows every process' vote and is able to decide properly.

- Consensus to the rescue. Now in an execution E^- in which some process crashes, or some message is delayed, P can propose a value to consensus (we say that P may *cons-propose* a value) and wait for the decision of the consensus. We first explain when P cons-proposes a value and then explain which value P cons-proposes. Now, at $2U$, if P receives at least one acknowledgement from a process in $\{P_1, P_2, \ldots, P_f\}$, then P cons-proposes a value immediately at $2U$. Otherwise, P asks $P_{f+1}, P_{f+2}, \ldots, P_n$ for the acknowledgements which $P_{f+1}, P_{f+2}, \ldots, P_n$ have received. To be more specific, if P is a process in $\{P_1, P_2, \ldots, P_f\}$, then P can always cons-proposes a value at $2U$ in E^-. If not and if at $2U$, P indeed receives no acknowledgement from any process in $\{P_1, P_2, \ldots, P_f\}$, then P eventually receives acknowledgement messages from $n-f$ out of n processes and then may cons-propose a value. At the point when P is ready to cons-proposes a value, P looks for every process' vote in the acknowledgement messages which P has received so far. If P finds that every process' vote is 1, then P cons-proposes 1; otherwise, P cons-proposes 0.

The state transition of P in E and in E^- is illustrated in Figure 1. We use there the following notations: *AND* denotes the logical AND of those 0's and 1's as votes; Y and N are the abbreviated for *yes* and *no* respectively; *self* denotes P, the process in question; *ack* denotes an acknowledgement; *cons* denotes consensus (which is not invoked if no process crashes and every message arrives in time).

Some remarks on the protocol are in order. Clearly, the strategy of decisions of our INBAC protocol is independent of the underlying consensus algorithm. In addition, INBAC encourages processes to propose 1 to consensus by looking at every process' vote in the acknowledgements received.

Best-case complexity. We now count the number of messages and that of message delays. Since in every nice execution every process decides at $2U$, then the number of message delays meets the lower bound (Theorem 1). As for the number of messages in any nice execution, at time 0, for every process P, f messages leave from P; at time $2U$, exactly f messages arrive at the same process P.[10] (This is because in INBAC, a backup process sends the acknowledgement of several votes $\mathcal{V}$ in one message). Therefore, among n processes, $2fn$ messages are exchanged in E, which meets the lower bound on the number of messages in Theorem 5. This optimal result shows that both lower bounds are tight, as summarized in Theorem 6.

In the version as described above, the complexity of INBAC of a failure-free execution in which some process votes 0 is the same as the complexity of any nice execution. We remark that our protocol INBAC may accelerate such failure-free execution by doing the following: if a process P votes 0, then P sends its vote to every process and decides 0 at the very beginning (and in the meantime, a process $Q \neq P$ who receives one vote of 0 decides 0 immediately). Then a failure-free execution in which some process votes 0 can terminate at the end of the *first* message delay, which is faster than any nice execution.

Theorem 6 (Message-optimal indulgent atomic commit given two message delays)**.** *Given any protocol that solves consensus in a network-failure system, INBAC solves indulgent atomic commit, and during every nice execution of INBAC, (a) any process decides after two message delays, and (b) n processes exchange $2fn$ messages.*

Proof sketch of correctness. (The full proof is deferred to Appendix B.) Obviously, every crash-failure execution of INBAC satisfies *validity.* Every such execution also satisfies *termination* because, only a process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ might wait, but such wait eventually terminates given that at most f processes might crash. Thus as long as a consensus protocol satisfies *termination* in any crash-failure system, INBAC also terminates in any crash-failure system.

Every execution of INBAC satisfies *agreement* given that consensus satisfies *agreement* and *validity.* We can show that by contradiction. If two processes decide differently, then the one which decides 1 must receive all the acknowledgements which it expects at $2U$ while the

[10] A message whose source and destination is the same does not need to be sent over the network; such a message arrives immediately and is not counted in the messages exchanged among the n processes.

one which decides 0 must have 0 as a decision of consensus. However, the one which decides 1 must have observed that (a) every process proposes 1, and (b) every process has f successful backups of its vote; thus as a result, no process can propose 0 to consensus. A contradiction.

Finally, as we claimed in the beginning of this section, we note a necessary design for the optimal protocol. We show in Lemma 6 that $f-1$ acknowledgements of other processes' votes are necessary. The proof of Lemma 5 is similar to the proof of Lemma 5, which is thus deferred to Appendix C.2. As both Lemma 5 and Lemma 6 are necessary in designing message-optimal protocols, e.g., given three message delays, they may be of independent interest and worth mentioning here.

Lemma 6 (Quick acknowledgements of other votes). *Let π be any indulgent atomic commit protocol. Let E be any nice execution of π. Let P decide at some time t_1 in E. Let process $R \neq P$. Let Θ be such set of processes that $\forall Q \in \Theta$, Q satisfies that before t_1 (including t_1) in E, R reaches Q, and subsequently Q reaches P. Among the messages whose destination is P, let $\mathcal{M}$ be the set of messages that arrives at P before or at t_1. For each $m \in \mathcal{M}$, let t_m be the time at which m leaves from its source and let $t_2 = \max_{m \in \mathcal{M}} t_m$.*

If $t_2 \leq 2U$, then $|\Theta| \geq f-1$.

6. RELATED WORK

6.1 Complexity of commit protocols

The study of atomic commit problems dates back to Skeen [16]. Later, substantial refinement [11, 14, 15] has been made, leading to the properties of NBAC we consider in this paper.

Complexity measures. We consider two measures of complexity: the classical notion of *number of messages*, and the *number of message delays*, following the complexity study by Lamport of consensus [22]. The use of this complexity measure (message delays) is justified by the general context of an arbitrary (asynchronous) system (considering network-failure executions) in [22] and in this paper. Unlike [17, 18], we do not consider the *number of steps* as a measure of time. In [17, 18], steps were defined for synchronous systems and do not fit a general asynchronous setting. (In addition, since steps and message delays measure time differently, even for the special case of synchronous NBAC, the results on number of steps in [17, 18] and our results on message delays are uncomparable.)

Complexity results. The most closely related works to ours are (a) Dwork and Skeen's lower bound on the number of messages [17, 25, 26] and (b) Charron-Bost and Schiper's bound on the number of *rounds* [39] (of which the tightness was shown by Dutta et al. [42]). Both works focus on synchronous NBAC, while our study is for an arbitrary (asynchronous) system as well as an arbitrary combination of properties of NBAC. For the special case of synchronous NBAC, we are the first to present a tight lower bound on both the number of messages and that of message delays.

Compared with previous work, we generalize Dwork and Skeen's necessary and sufficient number of messages when at most $n-1$ processes may crash among n processes to an arbitrary number of crashes. Still for the special case of synchronous NBAC, we make Charron-Bost and Schiper's lower bound on time complexity more precise. They showed a lower bound of two rounds. In their model, one round consists of one send phase and one receive phase [39, 43]. Thus a lower bound of two rounds only says that the number of send phases *or* receive phases is at least two: it does not articulate which one. Combined with our tight lower bound of one message delay, we get a clear picture of the time complexity of synchronous NBAC protocols: a process can decide at the earliest by the end of the first message delay, and if so, it has to send messages before its decision. In other words, for any synchronous NBAC protocol, before any process decides, two send phases and one receive phase are necessary. (The tight two-round protocol of [42] needs at least two message delays and thus does not help to get such a picture.)

To the best of our knowledge, the present paper is the first to study the complexity of atomic commit problems in a systematic way. For indulgent atomic commit, the most robust among atomic commit problems we study, no (non-trivial) lower bound on the number of message delays or the number of messages was known until this paper.[11] Table 4 summarizes the complexity results of previous work and the present paper. The number of message delays for previous work is left blank.

6.2 Commit protocols

Two-phase commit (2PC) [2] distinguishes one process as the leader, which is a single point of failure in the sense that if it crashes, every other process is blocking in the fear of disagreement [16]. To circumvent this, Skeen [16] proposed three-phase commit (3PC), which adds one message delay and $2n-2$ messages over 2PC, along with a termination protocol. However, as several papers [19, 21] pointed out, 3PC (as well as many of its variants) does not solve the potential conflict between two backup leaders at the same time given by the termination protocol in crash-failure executions. Gray and Lamport [21] proposed *PaxosCommit* based on Paxos consensus [29] to solve the disagreement of non-unique leaders in network-failure executions. They also proposed *faster PaxosCommit* [21], an optimization of PaxosCommit, removing one message delay.[12] Both Pax-

[11] Based on Charron-Bost and Schiper's two-round lower bound, Gray and Lamport [21] informally argued that two message delays should be optimal for indulgent atomic commit. However, by the model of rounds [39, 43], two rounds only imply a bound of one message delay.

[12] Gray and Lamport [21] pointed out a possible optimization (without details) for an atomic commit protocol, MD3PC, proposed in [20]. Then MD3PC achieves the same number of message delays and messages as faster PaxosCommit. As MD3PC and faster PaxosCommit are equally efficient in nice executions, MD3PC is omitted from the discussion.

Table 4: Complexity of Indulgent Atomic Commit, and Synchronous NBAC with f Crashes

	Indulgent atomic commit (this paper)	Sync. NBAC (this paper)	Sync. NBAC [17, 25, 26, 39, 42]
#delays	2	1	-
#messages	$2n - 2 + f$ (for $f \geq 2$)	$n - 1 + f$	$2n - 2$ (when $f = n - 1$) [17, 25, 26]

Table 5: Complexity of INBAC, (n-1+f)NBAC, 1NBAC, 2PC, PaxosCommit and faster PaxosCommit

	1NBAC (this paper)	(n-1+f)NBAC (this paper)	INBAC (this paper)	2PC [2]	Paxos-Commit [21]	Faster Paxos-Commit [21]
#delays	1	$2f + n - 1$	2	2	3	2
#messages	$n^2 - n$	$f + n - 1$	$2fn$	$2n - 2$	$nf + 2n - 2$	$2fn + 2n$ $-2f - 2$
Atomic commit	Sync. NBAC	Sync. NBAC	Indulgent	Blocking	Indulgent	Indulgent

osCommit [21] and faster PaxosCommit [21] solve indulgent atomic commit.

Table 5 summarizes the time and message complexity of our INBAC, our two optimal synchronous NBAC protocols: (n-1+f)NBAC and 1NBAC, 2PC, PaxosCommit, and faster PaxosCommit. To enable a fair comparison, we assume that each protocol starts when n processes send messages spontaneously.[13]

Clearly, our (n-1+f)NBAC and 1NBAC protocols are the best regarding messages and message delays respectively. Among indulgent atomic commit protocols, in the special case of $f = 1$, INBAC performs the best regarding both messages and message delays (for $n \geq 2$), and performs almost as efficiently as 2PC. Still among indulgent atomic commit protocols, PaxosCommit and our INBAC protocol show a tradeoff between time and message complexity: for $f \geq 2, n \geq 3$, PaxosCommit is better in messages while our INBAC protocol is better in message delays. On satisfaction of properties, our (n-1+f)NBAC and 1NBAC protocols and 2PC show a tradeoff between agreement and termination. 2PC guarantees agreement in an arbitrary (asynchronous) system (considering a network-failure execution) but not termination even if only crash failures are possible. On the other hand, (n-1+f)NBAC and 1NBAC terminate despite f crashes but an execution in an arbitrary (asynchronous) system may violate *agreement* (due to the use of noops for (n-1+f)NBAC and due to the optimal delay for 1NBAC respectively).

On a technical level, while solving the same problem, faster PaxosCommit and our INBAC protocol differ significantly in how they achieve two message delays: the design of INBAC follows immediately the proof of the lower bounds (Lemma 1 and Lemma 5) and does not invoke consensus in any nice execution, while faster PaxosCommit uses Paxos consensus in a non-black-box way in every execution.

6.3 Low-latency commit protocols with weak semantics

As observed in [44], 1-delay commit protocols proposed in [45, 46] assumes that all processes propose 1 before an execution starts. Jiménez-Peris et al. proposed a commit service which has the same latency as 2PC but allows a process to decide twice and differently. MDCC [47] proposed a variant of Paxos to coordinate transactions assuming all processes vote the same. Replicated Commit [48] executed also the Paxos protocol to commit transactions, assuming here that the votes from a majority of processes are already sufficient to commit. All these protocols solve different (and weaker) problems than classical atomic commit.

Calvin [49] eliminated the explicit commit protocol by using a deterministic locking scheme, using only one message to notify the decision; in fact, NBAC is only solved in failure-free executions where one message delay is (not surprisingly) sufficient. Helios [10] commits a distributed transaction if no conflict involving the transaction is detected across datacenters. Helios considers both failure-free and network-failure executions. In failure-free executions, optimal commit latency is achieved. In network-failure executions, the scheme proposed is far from the optimal in terms of complexity. Our INBAC protocol may be adapted to the needs of Helios with better complexity.

7. CONCLUDING REMARKS

We present the first systematic study of the (time and message) complexity of atomic commit. We give a collection of lower bounds and matching protocols, by which we also close many questions on atomic commit. Some questions remain open. For example, for the tradeoff between time and message complexity, the optimal number of messages given greater than two message delays for indulgent atomic commit is not yet clear (although we close the question for two message delays).

[13] Thus 1 delay from 2PC and 2 delays from PaxosCommit and faster PaxosCommit are removed respectively, while $n-1$ messages are removed from the three protocols respectively from their original counting.

8. REFERENCES

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[2] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, 1978, pp. 393–481.

[3] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, 1986.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 159–174, 2007.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[6] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI '10*, pp. 1–15.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

[8] J. Du, S. Elnikety, and W. Zwaenepoel, "Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks," in *SRDS '13*, pp. 173–184.

[9] M. K. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: Scalable sql storage for web applications," in *SOSP '15*, pp. 245–262.

[10] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Minimizing commit latency of transactions in geo-replicated data stores," in *SIGMOD '15*, pp. 1279–1294.

[11] V. Hadzilacos, "On the relationship between the atomic commitment and consensus problems," in *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, 1990, pp. 201–208.

[12] R. Guerraoui, "Revisiting the relationship between non-blocking atomic commitment and consensus," in *WDAG '95*, pp. 87–100.

[13] B. Charron-Bost, "Agreement problems in fault-tolerant distributed systems," in *SOFSEM '01*, pp. 10–32.

[14] R. Guerraoui, "Non-blocking atomic commit in asynchronous distributed systems with failure detectors," *Distrib. Comput.*, vol. 15, no. 1, pp. 17–25, 2002.

[15] R. Guerraoui, V. Hadzilacos, P. Kuznetsov, and S. Toueg, "The weakest failure detectors to solve quittable consensus and nonblocking atomic commit," *SIAM J. Comput.*, vol. 41, no. 6, pp. 1343–1379, 2012.

[16] D. Skeen, "Nonblocking commit protocols," in *SIGMOD '81*, pp. 133–142.

[17] C. Dwork and D. Skeen, "The inherent cost of nonblocking commitment," in *PODC '83*, pp. 1–11.

[18] K. V. S. Ramarao, "Complexity of distributed commit protocols," *Acta Informatica*, vol. 26, no. 6, pp. 577–595, 1989.

[19] I. Keidar and D. Dolev, "Increasing the resilience of atomic commit, at no additional cost," in *PODS '95*, pp. 245–254.

[20] R. Guerraoui, M. Larrea, and A. Schiper, "Reducing the cost for non-blocking in atomic commitment," in *ICDCS '96*, pp. 692–697.

[21] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, 2006.

[22] L. Lamport, "Lower bounds for asynchronous consensus," *Distrib. Comput.*, vol. 19, no. 2, pp. 104–125, 2006.

[23] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[25] V. Hadzilacos, "A knowledge-theoretic analysis of atomic commitment protocols," in *PODS '87*, pp. 129–134.

[26] O. Wolfson and A. Segall, "The communication complexity of atomic commitment and of gossiping," *SIAM J. Comput.*, vol. 20, no. 3, pp. 423–450, 1991.

[27] R. Guerraoui, "Indulgent algorithms (preliminary version)," in *PODC '00*, pp. 289–297.

[28] R. Guerraoui and N. Lynch, "A general characterization of indulgence," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 4, pp. 20:1–20:19, 2008.

[29] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[30] G. Taubenfeld, "Computing in the presence of timing failures," in *ICDCS '06*, pp. 16–16.

[31] P. Dutta and R. Guerraoui, "The inherent price of indulgence," *Distrib. Comput.*, vol. 18, no. 1, pp. 85–98, 2005.

[32] R. Guerraoui and M. Raynal, "The information structure of indulgent consensus," *IEEE Trans. Comput.*, vol. 53, no. 4, pp. 453–466, 2004.

[33] L. Sampaio and F. Brasileiro, "Adaptive indulgent consensus," in *DSN'05*, pp. 422–431.

[34] O. Bakr and I. Keidar, “Evaluating the running time of a communication round over the internet,” in *PODC ’02*, pp. 243–252.

[35] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[36] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[37] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[38] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[39] B. Charron-Bost and A. Schiper, “Uniform consensus is harder than consensus,” *J. Algorithms*, vol. 51, no. 1, pp. 15 – 37, 2004.

[40] R. D. Prisco, B. Lampson, and N. Lynch, “Revisiting the paxos algorithm,” *Theoretical Computer Science*, vol. 243, no. 1, pp. 35 – 91, 2000.

[41] R. Guerraoui and A. Schiper, “The generic consensus service,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 29–41, 2001.

[42] P. Dutta, R. Guerraoui, and B. Pochon, “Fast non-blocking atomic commit: an inherent trade-off,” *Inform. Process. Lett.*, vol. 91, no. 4, pp. 195 – 200, 2004.

[43] N. A. Lynch, *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

[44] M. Abdallah, R. Guerraoui, and P. Pucheral, “One-phase commit: does it make sense?” in *ICPADS ’98*, pp. 182–192.

[45] Y. J. Al-Houmaily and P. K. Chrysanthis, “An atomic commit protocol for gigabit-networked distributed database systems,” *J. Syst. Architect.*, vol. 46, no. 9, pp. 809 – 833, 2000.

[46] J. W. Stamos and F. Cristian, “A low-cost atomic commit protocol,” in *SRDS ’90*, pp. 66–75.

[47] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, “Mdcc: Multi-data center consistency,” in *EuroSys ’13*, pp. 113–126.

[48] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, “Low-latency multi-datacenter databases using replicated commit,” *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 661–672, 2013.

[49] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin: Fast distributed transactions for partitioned database systems,” in *SIGMOD ’12*, pp. 1–12.

[50] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

APPENDIX

A. INBAC

We describe here our INBAC protocol in detail. Some preliminary remarks are in order. (a) We assume that every process knows its own ID stored in the local variable i of that process. (b) We assume that a message delivery event has a higher priority than a timeout event; i.e., if both events occur at a process, the process is first triggered by the delivery event and then the timeout event. (c) Sometimes a process is triggered by both the delivery of some message m and a logical condition ℓ; we assume that if m arrives earlier than when ℓ is satisfied, then m (as well as the delivery of m) is queued to wait for the satisfaction of ℓ. (d) This protocol satisfies some properties in every crash-failure execution and thus we assume that one unit at the timer at every process is set to the known upper bound of the message delay of a given crash-failure system. (Clearly, in a network-failure execution of the protocol, message delays might violate the upper bound, and as a result, although the timer timeouts, a process does not receive the message which it sets the timer to wait for.) (e) The timer starts at time 0 when every process proposes its value. (f) Sending messages and processing messages are considered negligible in time. For the other protocols, we do not repeat these remarks when they still apply.

The pseudo code which we use to describe INBAC (and all the other protocols described in the later sections) follows the approach of Cachin et al. [50]. The pseudo code uses “callbacks”: an algorithm is described as a set of event handlers where a process reacts to incoming events by possibly triggering new events.

The communication channels are abstracted as a module called *PerfectPointToPointLinks*, denoted by *pl*. The module defines two events: <*pl*, Send|r, m> and <*pl*, Deliver|s, m>, where r is the receiver of the sending event, s is the sender of the message delivery event and m represents the message. The timer is abstracted as a module called *Timer*, denoted by *timer*. The module defines two events: <*timer*, Timeout> and **set** timer, where *timer* timeouts at the time set previously. A timer may be set several times at one process. The uniform consensus (defined in Definition 5, which terminates in every network-failure execution) is abstracted as a module called *IndulgentUniformConsensus*, denoted by *iuc*. The module defines two events: <*iuc*, Propose | v> and <*iuc*, Decide | d>, where v is the value proposed to *iuc* and d is the decision of *iuc*.

Finally, we also abstract indulgent atomic commit as a module called *IndulgentNonBlockingAtomicCommit*, denoted by *inbac*. The module defines two events (in addition to event <*inbac*, Init> which performs the initialization of the module once for all): <*inbac*, Propose | v> and <*inbac*, Decide | d> where v is the value proposed to indulgent atomic commit and d is the decision of indulgent atomic commit.

Note that the name of a module is considered as an identifier. The existence of multiple copies of the same module is possible in one algorithm. For example, later in protocol aNBAC, two timers are used and identified by different names.

```
Implements:
 IndulgentNonBlockingAtomicCommit, instance inbac.

Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.
  IndulgentUniformConsensus, instance iuc.

upon event <inbac, Init> do
  phase := 0;
  proposed := FALSE;
  decided := FALSE;
  collection0 := $\emptyset$;
  collection1 := $\emptyset$;
  collection_help := $\emptyset$;
  wait := FALSE;
  val := $\perp$;
  decision := $\perp$;
  proposal := $\perp$;
  cnt := 0;
  cnt_help := 0;

upon event <inbac, Propose | v> do
  val := v;
  forall $q \in \{P_1, \dots, P_f\}$ do
    trigger <pl, Send | q, [V, v]>;
  if $1 \le i \le f$ then
    trigger <pl, Send | $P_{f+1}$, [V, v]>;
  if $1 \le i \le f+1$ then
    set timer to 1;
  else
    set timer to 2;
    phase := 1;

upon event <pl, Deliver | p, [V, v]> and phase = 0 do
  collection0 := collection0 $\cup \{(p, v)\}$;

upon event <timer, Timeout> and phase = 0 and $1 \le i \le f$ do
  forall $q \in \Omega$ do
    trigger <pl, Send | q, [C, collection0]>;
  phase := 1;
  set timer to 2;

upon event <timer, Timeout> and phase = 0 and i = f + 1 do
  forall $q \in \{P_1, P_2, \dots, P_f\}$ do
    trigger <pl, Send | q, [C, collection0]>;
  phase := 1;
  set timer to 2;

upon event <pl, Deliver | p, [C, collection]> do
  collection1 := collection1 $\cup \{(p, collection)\}$;
  cnt := cnt + 1;

upon event <timer, Timeout> and phase = 1 and not decided and not proposed and $i \ge f + 1$ do
  phase := 2;
  collection_val := $\bigcup_{(p,c)\in\ collection1} c$;
 collection0 := collection0 $\cup$ collection_val $\cup \{(self, val)\}$;
 if collection1 $= \{(P_j, c_j) \mid 1 \le j \le f\}$ where $c_j = \{(P_k, val_k) \mid 1 \le k \le n\}$ for every $j$, $1 \le j \le f$ (with $val_k$ being the proposal of $P_k$) then
    decision := $\text{AND}_{1\le k \le n} val_k$;
    decided := TRUE;
    trigger <inbac, Decide | decision>;
  else if $cnt \ge 1$ then
   if for every process $P_k$, $1 \le k \le n$, $\exists\, val_k$ s.t. $(P_k, val_k) \in \bigcup_{(p,c)\in\ collection1} c$ then
      proposal := $\text{AND}_{1\le k\le n} val_k$;
      proposed := TRUE;
      trigger <iuc, Propose | proposal>;
    else
      proposed := TRUE;
      trigger <iuc, Propose | 0>;
  else
    wait := TRUE;
    forall $q \in \{P_{f+1}, P_{f+2}, \dots, P_n\}$ do
      trigger < pl, Send | q, [HELP]>;

upon event <pl, Deliver | p, [HELP]> and phase = 2 and $i \ge f+1$ do
  trigger < pl, Send | p, [HELPED, collection0]>;

upon event <pl, Deliver | p, [HELPED, collection]> and $i \ge f+1$ do
  collection_help := collection_help $\cup$ collection;
  cnt_help := cnt_help + 1;

upon $cnt + cnt\_help \ge n - f$ and wait and not proposed and not decided and $i \ge f+1$ do
  wait := FALSE;
 if collection1 $= \{(P_j, c_j) \mid 1 \le j \le f\}$ where $c_j = \{(P_k, val_k) \mid 1 \le k \le n\}$ for every $j$, $1 \le j \le f$ (with $val_k$ being the proposal of $P_k$) then
    decision := $\text{AND}_{1\le k\le n} val_k$;
    decided := TRUE;
    trigger <inbac, Decide | decision>;
  else if $cnt \ge 1$ then
   if for every process $P_k$, $1 \le k \le n$, $\exists\, val_k$ s.t. $(P_k, \text{val}_k) \in \bigcup_{(p,c)\in\ collection1} c$ then
      proposal := $\text{AND}_{1\le k\le n} val_k$;
      proposed := TRUE;
      trigger <iuc, Propose | proposal>;
    else
      proposed := TRUE;
      trigger <iuc, Propose | 0>;
  else
   if collection_help $= \{(P_k, val_k) \mid 1 \le k \le n\}$ where $val_k$ is the proposal of $P_k$ then
      proposal := $\text{AND}_{1\le k\le n} val_k$;
      proposed := TRUE;
      trigger <iuc, Propose | proposal>;
    else
      proposed := TRUE;
      trigger <iuc, Propose |0 >;
```

```
upon event <timer, Timeout> and $phase = 1$ and not decided and not proposed and $1 \leq i \leq f$ do
    if $collection1 = \{(P_j, c_j) \mid 1 \leq j \leq f+1\}$ where $c_j = \{(P_k, val_k) \mid 1 \leq k \leq n\}$ for every $j$, $1 \leq j \leq f$ and $c_{f+1} = \{(P_k, val_k) \mid 1 \leq k \leq f\}$
    (with $val_k$ being the proposal of $P_k$) then
        decision := $\text{AND}_{1 \leq k \leq n} val_k$;
        decided := TRUE;
        trigger <inbac, Decide | decision>;
        return;
    if for every process $P_k$, $1 \leq k \leq n$, $\exists\, val_k$ s.t. $(P_k, val_k) \in \bigcup_{(p,c) \in collection1} c$ then
        proposal := $\text{AND}_{1 \leq k \leq n} val_k$;
        proposed := TRUE;
        trigger <iuc, Propose | proposal>;
    else
        proposed := TRUE;
        trigger <iuc, Propose | 0>;

upon event <iuc, Decide | v> and not decided do
    decided := TRUE;
    trigger <inbac, Decide | v>;
```

B. CORRECTNESS OF INBAC

Proof. (Proof of Theorem 6.) First, we prove that every execution of INBAC satisfies the *agreement* property.

Agreement. By contradiction. Suppose that in some execution E, two different processes P and Q decide differently. Suppose further that P decides 1 and Q decides 0. Given that consensus satisfies the *agreement* property, at least one of P and Q's decisions is *not* a result of the decision of the consensus.

If neither of P and Q's decisions is a result of the decision of the consensus, then either process decides the value of its local variable *decision*. Since *decision* is assigned as the AND of the n processes' votes to *inbac* at every process, P and Q must agree on their decisions, which contradicts our assumption. If P's decision is a result of the decision of the consensus, then by the *validity* property of consensus, some process R proposes 1 to *iuc*. Therefore R's local variable *proposal* is 1, which is equal to the AND of the n processes' votes to *inbac*. Now that Q's decision is equal to its local variable *decision*, which is the AND of the n processes' votes to *inbac*, P and Q must agree on their decisions, which contradicts our assumption.

As a result, Q's decision must be a result of the decision of the consensus while P's decision must not be. Now P's local variable *decision* is 1. Therefore, every process proposes 1 to *inbac* and at the same time, if any process assigns a value to its local variable *proposal* or *decision*, it can only assign a 1. Since Q's decision is a result of the consensus, by the *validity* property of consensus, some process R (not necessarily Q) proposes 0 to consensus. First, we assume that $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$ and examine whether R exists. As P decides 1, variable *collection*0 at every process in $\{P_1, P_2, \ldots, P_f\}$ is $\{(P_k, val_k) | 1 \leq k \leq n\}$. Therefore, $R \notin \{P_1, P_2, \ldots, P_f\}$. I.e., $R \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$. Then variable *cnt* at R must be 0 and thus for R to propose 0, R must have *cnt_help* $= n - f$, i.e., every process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ has sent to R their variable *collection*0. As a result, P has also sent its *collection*0, which is updated to $\{(P_k, val_k) | 1 \leq k \leq n\}$ when *phase* $= 2$. This leads R to propose 1 to consensus. A contradiction.

Now we assume that $P \in \{P_1, P_2, \ldots, P_f\}$ and examine whether R exists. Similarly, variable *collection*0 at every process in $\{P_1, P_2, \ldots, P_f\}$ is $\{(P_k, val_k) | 1 \leq k \leq n\}$. Moreover, variable *collection*0 at P_{f+1} includes $\{(P_k, val_k) | 1 \leq k \leq f\}$ as a subset. Again, R must belong to $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$. For R to propose 0, R must have *cnt_help* $= n - f$, i.e., every process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ has sent to R their updated variable *collection*0, the union of which is equal to $\{(P_k, val_k) | 1 \leq k \leq n\}$. (Variable *collection*0 at every process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ is updated to include its own vote.) This again leads R to propose 1 to consensus. A contradiction.

Next, we prove that every network-failure execution of INBAC satisfies the *validity* property, and the *termination* property.

Validity. Clearly, the *validity* property can be separated into the *commit-validity* property: if a process decides 1, then every process proposes 1; and the *abort-validity* property: if a process decides 0, then some process proposes 0 or a failure occurs. The proof here (and the proofs for the correctness of protocols later) proves that the protocol satisfies the *commit-validity* property and the *abort-validity* property respectively.

Commit-Validity. Suppose that some process P decides 1. If P's decision is a result of the decision of the consensus, then since consensus satisfies the *validity* property, some process R (not necessarily P) must propose 1 to consensus. Since variable *proposal* at R is equal to the AND of the n votes, every process proposes 1 to *inbac*. If P's decision is not a result, then variable *decision* at P is equal to the AND of the n votes, which implies that every process proposes 1 to *inbac*.

Abort-Validity. Suppose that process P decides 0. If P's decision is equal to variable *decision* at P or variable *proposal* at some other process R, then some process must propose 0 to *inbac*. If not, then some process R (not necessarily P) must have proposed 0 to consensus in the case where some value is missing in variable *collection_help* or the collection $\bigcup_{(p,c) \in collection1} c$ at R. This indicates that some message does not arrive before the timer issues a timeout event, which is set to the upper bound of the message delay. Then, in a network-failure system, we can safely conclude that a failure occurs. Thus the *abort-validity* property is satisfied.

Termination. By contradiction. Suppose that some correct process P does not decide. P assigns *phase* to 1 in finite time. Then P is triggered by the event that the timer issues a timeout and *phase* = 1, when P has not proposed to consensus or decided in *inbac*. If $P \in \{P_1, P_2, \ldots, P_f\}$, then since consensus *iuc* satisfies the *termination* property in a network-failure system, P eventually decides in *inbac*. A contradiction. If $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$, then P assigns *phase* to 2 in finite time. In fact, every correct process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ assigns *phase* to 2 in finite time. Since P does not decide, thus by the *termination* property of *iuc* in a network-failure system, P must assign *wait* to TRUE and wait for the condition $cnt + cnt\_help \geq n - f$ to satisfy. If the condition is satisfied and the corresponding event is triggered, then P eventually decides in *inbac*. In other words, for P to not decide, the condition should never be satisfied.

However, when *wait* is assigned to TRUE, *cnt* is 0. Only the message of [C, *] increments *cnt*. Since $P \in \{P_{f+1}, P_{f+2}, \ldots, P_n\}$, then the message of [C, *] that arrives at P can only be from a process in $\{P_1, P_2, \ldots, P_f\}$, each correct process of which must send [C, *] to P. On the other hand, *cnt_help* at P is incremented if a message from a process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ arrives. Every correct process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ also must send message [HELPED, *] to P. As at most f processes can crash and messages eventually arrive at their destinations respectively, *cnt* + *cnt_help* is eventually equal to or greater than $n - f$. In other words, the condition is eventually satisfied. A contradiction.

Finally, since consensus satisfies the *termination* property in an network-failure system (assuming a majority of correct processes), INBAC also satisfies the *termination* property in an network-failure system (assuming a majority of correct processes).

Therefore, given that consensus can be implemented for a network-failure system, protocol INBAC (i.e., instance *inbac*) solves indulgent atomic commit. □

C. PROOFS OF LEMMAS

C.1 Proof of Lemma 5

By contradiction. Suppose that $|\Theta| \leq f - 1$. Denote by Φ the set of P and the processes which P has reached at t_2. For each process $Q \in \Theta$, denote by τ_Q the time at which P reaches Q in E. For each process $Q^- \in \Phi \backslash (\Theta \cup \{P\})$, denote by τ_{Q^-} the time at which P reaches Q^- in E.

We build a crash-failure execution E_0 based on E. In E_0, P crashes before sending any message (i.e., P crashes at time 0). For Q, E_0 is the same as E until Q crashes and Q crashes before sending any message that is expected to send upon the message(s) received by Q at τ_Q (i.e., Q crashes at τ_Q). For every other process R, E_0 is the same as E until some process in $\Theta \cup \{P\}$ timeouts at some process not in $\Theta \cup \{P\}$.

Now we construct E_0 after some process timeouts as follows. First, we consider the earliest timeout. The earliest timeout occurs at a process in $\Phi \backslash (\Theta \cup \{P\})$. (By Lemma 1, $|\Phi \backslash \{P\}| \geq f$. As $|\Theta| \leq f - 1$, $\Phi \backslash (\Theta \cup \{P\})$ is non-empty.) Let $Q^- \in \Phi \backslash (\Theta \cup \{P\})$ be the process at which the earliest timeout occurs. Denote by t_3 at which the earliest timeout occurs. Clearly, $t_3 > U$. If Q^- sends any message m_1 upon the timeout event, then we assume that m_1 arrives at its destination at time $t_3 + U$. Second, any other message that is different from E due to the timeout events arrives in a delay similarly, i.e., with the same message delay U. Finally, every message that is sent after t_2 arrives later than t_1. Moreover, in E_0, P proposes 0, every other process proposes 1 and no process in $\Omega \backslash (\Theta \cup \{P\})$ crashes. As $|\Theta| \leq f - 1$ and $t_1 - t_2 \leq U$, E_0 is a legitimate crash-failure execution of π. Any remaining process $R \in \Omega \backslash (\Theta \cup \{P\})$ decides 0 in E_0. W.l.o.g., let R be the earliest process that decides. Denote by t_4 the time at which R decides.

Then based on E and E_0, we build a network-failure execution E_{async}. In E_{async}, every process proposes 1 and no process crashes. Therefore, E_{async} starts as E. Then we construct E_{async} such that:

- Every message from P to a process in $\Omega \backslash (\Theta \cup \{P\})$ arrives later than $\max(t_1, t_4)$;
- Every message from Q to a process in $\Omega \backslash (\Theta \cup \{P\})$ sent after or at τ_Q arrives later than $\max(t_1, t_4)$;
- Every message from a process in $\Phi \backslash (\Theta \cup \{P\})$ to P arrives later than $\max(t_1, t_4)$;
- Every message from a process Q^- in $\Phi \backslash (\Theta \cup \{P\})$ to Q sent after or at τ_{Q^-} arrives later than $\max(t_1, t_4)$.
- Every message sent after t_2 arrives later than t_1.

In addition, the rest of the messages which are communicated among $\Omega \backslash (\Theta \cup \{P\})$ are (sent/received) the same as E_0 after the first timeout event. This first timeout event occurs at the same time t_3 at Q^- in both E_{async} and E_0. Process Q^- might send some message m_1 due to the timeout event. (If Q^- is not a process in $\Phi \backslash (\Theta \cup \{P\})$, then m_1 arrives later than t_1 according to the construction, which makes certainly P's decision the same as in E.)

If m_1 is sent to P or Q, then we assume that m_1 arrives later than t_1; if m_1 is sent to some process O in $\Omega \backslash (\Theta \cup \{P\})$, then m_1 arrives at $t_3 + U$ and thus O can only send some message m_2 (due to m_1) after or at $t_3 + U$. As $t_3 > U$, $t_3 + U > 2U \geq t_2$ and therefore, m_2 also arrives later than t_1. Thus any message that is different from E due to the timeout events arrives later than t_1. Then E_{async} and E are indistinguishable for P before and at t_1. As a result, P decides 1 at t_1.

Process R is among $\Omega \backslash (\Theta \cup \{P\})$. For R, E_{async} is the same as E_0 before and at t_4. As a result, R decides 0 at t_4.

Clearly, E_{async} is a network-failure execution of π that does not satisfy the *agreement* property. A contradiction to the assumption that π solves indulgent atomic commit.

C.2 Proof of Lemma 6

By contradiction. Suppose that $|\Theta| \leq f - 2$. Denote by τ_Q the time at which R reaches Q in E. Denote by τ_P the time at which R reaches P in E. Denote by Φ

the set of R and the processes which R has reached at t_2. Denote by τ_{Q^-} the time at which R reaches Q^- for each process $Q^- \in \Phi(\Theta \cup \{P, R\})$ in E.

We build a crash-failure execution E_0 based on E. In E_0, R crashes before sending any message (i.e., R crashes at time 0). For Q, E_0 is the same as E until Q crashes and Q crashes before sending any message that is expected to send upon the message(s) received by Q at τ_Q (i.e., Q crashes at τ_Q). For P, E_0 is the same as E until P crashes before sending any message that is expected to send upon the message(s) received by P (i.e., P crashes at τ_P). For every other process O, E_0 is the same as E until some process in $\Theta \cup \{P, R\}$ timeouts at some process not in $\Theta \cup \{P, R\}$.

Now we construct E_0 after some process timeouts as follows. First, we consider the earliest timeout. If $\Phi \backslash (\Theta \cup \{P, R\})$ is empty, the earliest timeout occurs at a process later than t_2. If $\Phi \backslash (\Theta \cup \{P, R\})$ is non-empty, the earliest timeout occurs at a process in $\Phi \backslash (\Theta \cup \{P, R\})$. Let Q^- be the process at which the earliest timeout occurs. Denote by t_3 at which the earliest timeout occurs. Certainly, whether $Q^- \in \Phi \backslash (\Theta \cup \{P, R\})$ or not, $t_3 > U$. W.l.o.g., we assume that $Q^- \in \Phi \backslash (\Theta \cup \{P, R\})$. If Q^- sends any message m_1 upon the timeout event, then we assume that m_1 arrives at its destination at time $t_3 + U$. Second, any other message that is different from E due to the timeout events arrives in a delay similarly, i.e., with the same message delay U. Finally, every message that is sent after t_2 arrives later than t_1.

Moreover, in E_0, R proposes 0, every other process proposes 1 and no process in $\Omega \backslash (\Theta \cup \{P, R\})$ crashes. As $|\Theta| \leq f - 2$ and $t_1 - t_2 \leq U$, E_0 is a legitimate crash-failure execution of π. Any remaining process $O \in \Omega \backslash (\Theta \cup \{P, R\})$ decides 0 in E_0. W.l.o.g., let O be the earliest process that decides. Denote by t_4 at which O decides.

Then based on E and E_0, we build a network-failure execution E_{async}. In E_{async}, every process proposes 1 and no process crashes. Therefore, E_{async} starts as E. Let $\Omega_1 = \Omega \backslash (\Theta \cup \{P, R\})$. Let $\Phi_1 = \Phi \backslash (\Theta \cup \{P, R\})$. Then we construct E_{async} such that:

- Every message from R to a process in Ω_1 arrives later than $\max(t_1, t_4)$;
- Every message from Q to a process in Ω_1 sent after or at τ_Q arrives later than $\max(t_1, t_4)$;
- Every message from P to a process in Ω_1 sent after or τ_P arrives later than $\max(t_1, t_4)$.
- Every message from a process in Φ_1 to R arrives later than $\max(t_1, t_4)$;
- Every message from a process Q^- in Φ_1 to Q sent after or at τ_{Q^-} arrives later than $\max(t_1, t_4)$.
- Every message from a process Q^- in Φ_1 to P sent after or at τ_{Q^-} arrives later than $\max(t_1, t_4)$.
- Every message sent after t_2 arrives later than t_1.

In addition, the rest of the messages which are communicated among $\Omega \backslash (\Theta \cup \{P, R\})$ are (sent/received) the same as in E_0 after the first timeout event. This timeout event occurs at the same time t_3 at Q^- in both E_{async} and E_0.

Process Q^- might send some message m_1 due to the timeout event. If m_1 is sent to P, Q or R, then we assume that m_1 arrives later than t_1; if m_1 is sent to some process O in $\Omega \backslash (\Theta \cup \{P, R\})$, then O can only send some message m_2 after or at $t_3 + U$. As $t_3 > U$, $t_3 + U > 2U \geq t_2$ and therefore, m_2 also arrives later than t_1. Thus any message that is different from E due to the timeout events arrives later than t_1. Then E_{async} and E are indistinguishable for P before and at t_1. As a result, P decides 1 at t_1.

Process O is among $\Omega \backslash (\Theta \cup \{P, R\})$. For O, E_{async} is the same as E_0 before and at t_4. As a result, O decides 0 at t_4.

Clearly, E_{async} is a network-failure execution of π that does not satisfy the *agreement* property. A contradiction to the assumption that π solves indulgent atomic commit.

D. DELAY-OPTIMAL PROTOCOL

In this section, we present 1NBAC that (a) solves NBAC in every crash-failure execution, (b) satisfies validity and termination in every network-failure execution and (c) decides in one message delay in every nice execution.

```
Implements:
    1NonBlockingAtomicCommit, instance 1nbac.

Uses:
    PerfectPointToPointLinks, instance pl.
    Timer, instance timer.
    UniformConsensus, instance uc.

upon event <1nbac, Init> do
    phase := 0;
    proposed := FALSE;
    decided := FALSE;
    decision := $\perp$;
    collection0 := $\emptyset$;
    collection1 := $\emptyset$;

upon event <1nbac, Propose | v> do
    decision := v;
    forall q $\in$ $\Omega$ do
        trigger <pl, Send | q, [V, v]>;
    set timer to 1;

upon event <pl, Deliver | p, [V, v]> do
    collection0 := collection0 $\cup$ {p};
    decision := decision AND v;

upon event <timer, Timeout> and phase = 0 do
    if collection0 = $\Omega$ then
        forall q $\in$ $\Omega$ do
            trigger <pl, Send | q, [D, decision]>;
        if not decided then
            decided := TRUE;
        trigger <1nbac, Decide | decision>;
    else
        phase := 1;
        set timer to 2;

upon event <pl, Deliver | p, [D, d]> do
    collection1 := collection1 $\cup$ {p};
    decision := d;

upon event <timer, Timeout> and $phase = 1$ do
    if not decided then
        if $collection1 = \emptyset$ then
            decision := 0;
        proposed := TRUE;
        trigger <uc, Propose | decision>;

upon event <uc, Decide | d> do
    if not decided then
        decided := TRUE;
        trigger <1nbac, Decide | d>;
```

Proof. (Proof of correctness of 1NBAC.)
Termination. Every correct process proposes a value and sets a timer when $phase = 0$. When the timer timeouts, every correct process either decides, or sets again the timer and assigns $phase = 1$. When the timer timeouts again, the correct process proposes a value to *uc*. Thus, by the *termination* property of consensus, every correct process decides.

Commit-Validity. If process P decides 1, then by the *validity* property of consensus and the protocol itself, there exists process Q (not necessarily P) who sends [D, 1] in phase 0 and therefore every process proposes 1. In other words, a failure occurs. Thus, the *commit-validity* property is satisfied.

Abort-Validity. If process P decides 0, then either some process P decides 0 in phase 0, which implies that some process proposes 0, or by the *validity* property of consensus, some process Q proposes 0 to *uc* in phase 1, which implies that Q receives fewer than n messages in phase 0. Thus, the *abort-validity* property is satisfied.

Agreement. By contradiction. Suppose that two different processes P and Q decide 1 and 0 respectively, in a crash-failure execution. Then by the *commit-validity* property and the *abort-validity* property, every process proposes 1 and some process crashes before Q decides. By the *agreement* property of consensus, P and Q cannot both follow the decision of *uc* to decide. Thus P decides 1 in phase 0 and Q decides 0 as a decision of *uc*.

Since P decides in phase 0, P succeeds in sending $[D, 1]$ to every other process. Moreover, since every process proposes 1 to 1*nbac*, no process sends $[D, 0]$ after the first message delay. Thus thanks to the synchronous communication, every process that has not decided yet receives $[D, 1]$ and proposes 1 to *uc*. Thus by the *validity* property of consensus, Q cannot decide 0 as a decision of *uc*. A contradiction. □

E. MESSAGE-OPTIMAL PROTOCOLS

E.1 0NBAC

Here we present our 0NBAC protocol. For 0NBAC, every failure-free execution solves NBAC, every network-failure execution satisfies agreement and termination, and n processes exchange 0 message in every nice execution.

```
Implements:
    0MessageAtomicCommit, instance 0nbac.

Uses:
    PerfectPointToPointLinks, instance pl.
    UniformConsensus, instance uc.
    Timer, instance timer.

upon event <0nbac, Init> do
    myvote := $\perp$;
    myack := $\emptyset$;
    decided := FALSE;
    zero := FALSE;
    phase := 0;

upon event <0nbac, Propose | v> do
    myvote := v;
    if $v = 0$ then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, [V, 0]>;
    set timer to time 1;
    phase := 1;

upon event <pl, Deliver | p, [V, v]> and $phase = 1$ do
    zero := TRUE;
    trigger <pl, Send | p, [ACK]>;

upon event <pl, Deliver | p, [B, b]> and $phase = 2$ do
    if not ($myvote = 1$ and decided) then
        trigger <pl, Send | p, [ACK]>;

upon event <pl, Deliver | p, [ACK]> do
    myack := myack $\cup$ {p};

upon event <timer, Timeout> and $phase = 1$ do
    $phase = 2$;
    if zero = FALSE and $myvote = 1$ then
        decided := TRUE;
        trigger <0nbac, Decide | 1>;
    else if zero = TRUE and $myvote = 1$ then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, [B, 0]>;
        set timer to time 3;
    else
        set timer to time 2;

upon event <timer, Timeout> and $phase = 2$ do
    if $myack \subset \Omega$ then
        trigger <uc, Propose | 1>;
    else
        trigger <uc, Propose | 0>;
```

```
upon event <uc, Decide | d> and not decided do
    trigger <0nbac, Decide | d>;
    decided := TRUE;
```

Proof. (Proof of correctness of 0NBAC.)
Termination. Every correct process P proposes a vote v and sets *timer* to 1. Then when *timer* first timeouts, P either decides, or again sets *timer*. At the second timeout of *timer*, every correct process (which has not yet decided) proposes to *uc*, which eventually decides by the *termination* property of *uc* in a network-failure execution.

Commit-Validity. We only need to prove validity in every failure-free execution. If in a failure-free execution, a process P decides 1, then either P decides 1 at the first timeout or P decides the decision of consensus *uc*. If P decides at the first timeout, then P does not receive any message [V, 0], which implies that every process proposes 1. If P decides the decision of *uc*, then some process Q proposes 1 at Q's second timeout. Either Q's vote is 0 or Q receives a message [V, 0] before the first timeout. In either case, message [V, 0] is sent to all process when the local variable *phase* at every process is 1. Then in a failure-free execution, no process decides at the first timeout. However, for Q to propose 1, again in a failure-free execution, there must be some process R such that R's vote is 1 and R has decided at the first timeout, which leads to a contradiction. Therefore P cannot decide the decision of *uc* in a failure-free execution. As a result, every process proposes 1, which satisfies *commit-validity.*

Abort-Validity. We only need to prove validity in every failure-free execution. If a process P decides 0, then some process Q has proposed 0 to *uc*. Then Q has votes 0 or has received a vote of 0, which satisfies the *abort-validity* property.

Agreement. By contradiction. Suppose that E is a network-failure execution in which two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. Thus Q's decision must be a decision of *uc* while P's decision is not. Then some process R must have proposed 0 to *uc*. As a result, R has *timer* timeout twice at itself. Suppose that the vote of R to the commit protocol is 0. Then the second timeout is at time 2. We argue that R cannot receive P's acknowledgement of R's message $[V, 0]$ at the second timeout. If so, P's local variable *zero* turns true before P's first timeout, which contradicts to P's decision of 1. Thus, the vote of R to the commit protocol can only be 1, and R's second timeout is at time 3. Similarly, we argue that R cannot receive P's acknowledgement of R's message $[B, 0]$ at the second timeout. If so, P's local variables *phase* = 2 and *decided* = FALSE must hold when P sends the acknowledgment, which again contradicts to P's decision of 1 (without invoking *uc*). □

E.2 Message-optimal protocol for synchronous NBAC: (n-1+f)NBAC

We start with a notation convention: symbol % represents modulo in the protocol except that if the remainder is 0, the result of % is n instead of 0. We also use symbol % in the next protocol, where we do not repeat the notation convention. We change the timer slightly from the other protocols: the timer here starts at time 1 when the first sending event happens.

```
Implements:
    NonBlockingAtomicCommit, instance nbac.

Uses:
    PerfectPointToPointLinks, instance pl.
    Timer, instance timer.

upon event <nbac, Init> do
    decision := ⊥;
    decided := FALSE;
    delivered := FALSE;
    phase := 0;

upon event <nbac, Propose | v> do
    decision := v;
    if $i = 1$ then
        trigger <pl, Send | $P_2$, decision>;
    if $i = 1$ then
        set timer to time $n + 1$;
        phase := 2;
    else
        set timer to time $i$;
        phase := 1;

upon event <pl, Deliver | p, v> do
    decision := decision AND v;
    if phase $\leq 2$ then
        if $p = P_{(i-1)\%n}$ then
            delivered := TRUE;
    else if not decided then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, decision>;

upon event <timer, Timeout> and phase = 1 do
    if delivered = FALSE then
        decision := 0;
    if decision = 1 then
        trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
    else if $i = n$ then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, decision>;
    delivered := FALSE;
    if $i \geq f + 1$ then
        set timer to time $n + 2f + 1$;
        phase := 3;
    else
        set timer to time $n + i$;
        phase := 2;
```

```
upon event <timer, Timeout> and phase = 2 do
  if delivered = FALSE then
    decision := 0;
  if decision = 1 and $i \neq f$ then
    trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
  if decision = 0 then
    forall $q \in \Omega$ do
      trigger <pl, Send | q, decision>;
  delivered := FALSE;
  set timer to time $n + 2f + 1$;
  phase := 3;

upon event <timer, Timeout> and phase = 3 do
  decided := TRUE;
  trigger <nbac, Decide | decision>;
```

Proof. (Proof of correctness of (n-1+f)NBAC.)
Termination. When a process proposes a value or its local *timer* timeouts, it assigns a value to *phase*. Each time a process assigns a value to *phase*, it sets a timer. Since every correct process proposes a value, then every correct process enters phase 3 and has *timer* timeout at $n+2f+1$. Every correct process decides at $n+2f+1$.

Commit-Validity. We only need to prove validity in every crash-failure execution. If process P decides 1, then at time $n+2f+1$, P's local variable *decision* $= 1$. This leads to three facts: (a) that P has received no 0 from other processes; (b) that P's local variable *delivered* is TRUE when the timeout event for phase 1 (and if P is among $P_1, P_2, \ldots, P_f$, P's local variable *delivered* is TRUE when the timeout event for phase 2 occur at P); and (c) that P proposes 1. If P is among $P_1, P_2, \ldots, P_f$, then according to (b), P has received 1 at phase 2, which implies that every process proposes 1. If P is P_n, then according to (b), P has received 1 at phase 1, which implies that every process proposes 1. If P is not among $P_1, P_2, \ldots, P_f, P_n$, then according to (a), P does not receive 0 from $P_n, P_1, \ldots, P_f$ at time $n+1, \ldots, n+f+1$ respectively. Since at most f processes can crash, one process Q among $P_n, P_1, \ldots, P_f$ is instructed by the protocol to not send 0 to P. This implies that Q has received 1 at phase 2 if Q is among $P_1, P_2, \ldots, P_f$ or Q has received 1 at phase 1 if Q is P_n. Therefore, every process proposes 1.

Abort-Validity. We only need to prove validity in every crash-failure execution. If process P decides 0, then P's local variable *decision* $= 0$. Then either (a) P has proposed $v = 0$, or (b) P has received 0 from other processes, or (c) P's local variable *delivered* is FALSE when the timeout event for phase 1 or the timeout event for phase 2 occurs at P.

If P receives 0 from another process Q, then since a process only sends its local variable *decision* to other processes (if it sends any message), w.l.o.g., we may assume that Q is the earliest process that has local variable *decision* $= 0$. As a result, either Q has proposed $v = 0$, or Q's local variable *delivered* is FALSE when the timeout event for phase 1 or the timeout event for phase 2 occurs at Q.

Then, to examine the *abort-validity* property, we need only to examine the case where *delivered* is FALSE for Q, and case (c) for P. Let P_i be either process. As *delivered* is FALSE, P_i does not receive any message from $P_{(i-1)\%n}$ before the timeout event for phase 1 or the timeout event for phase 2 occurs. At the same time, for $2 \leq i \leq n$, $P_{(i-1)\%n}$ is instructed by the protocol to send a message to $P_{i\%n}$ in phase 1 if $P_1, P_2, \ldots, P_{i-2}$ do not crash and $P_1, P_2, \ldots, P_{i-1}$ propose 1; for $i = 1$, $P_{(i-1)\%n}$ is instructed by the protocol to send a message to $P_{i\%n}$ in phase 2; and for $2 \leq i \leq f$, $P_{(i-1)\%n}$ is instructed by the protocol to send a message to $P_{i\%n}$ in phase 2. As *delivered* is FALSE, then some process crashes or some process proposes 0.

In conclusion, the *abort-validity* property is satisfied.

Agreement. By contradiction. Suppose that two different processes P and Q decide 1 and 0 respectively in a crash-failure execution. Then by the *commit-validity* property and the *abort-validity* property, every process proposes 1 and some process crashes before Q decides.

Since Q decides 0, then Q's local variable *decision* is assigned to 0 at some point in phase 1, phase 2, or phase 3. Suppose that Q assigns *decision* to 0 in phase 1. If $Q \neq P_n$, then Q refuses to send a message, which would lead P to decide 0; if $Q = P_n$, then Q sends 0 to P, which would also lead P to decide 0. A contradiction. Suppose that Q assigns *decision* to 0 in phase 2, then Q also sends 0 to P, which would again lead P to decide 0. A contradiction.

Suppose that Q assigns *decision* to 0 in phase 3, then Q only does the assignment at time $n+2f+1$ or later. Otherwise, since both P and Q are alive at $n+2f+1$, when Q sends *decision* to P after the assignment, then the network could schedule the message so that P receives 0 before time $n+2f+1$ and decide 0 at time $n+2f+1$. Now that Q does the assignment at time $n+2f+1$, some process must send 0 to Q at time $n+2f$ or later. In fact, in order for Q to receive 0, between time $n+f$ and time $n+2f$, there must be at least $f+1$ process that try to send 0 to every process. However, those processes all fail to send 0 to P. This gives a contradiction: $f+1$ processes must have crashed (to make all those attempts fail) while at most f processes may crash. □

E.3 aNBAC

Similar to (n-1+f)NBAC, the timer is slightly changed: the timer here starts at time 1 when the first sending event happens.

Implements:
ANonBlockingAtomicCommit, **instance** *anbac*.

Uses:
PerfectPointToPointLinks, **instance** *pl*.
Timer, **instance** *timer*.
Timer, **instance** *timer0*.

```
upon event <anbac, Init> do
    decision := $\bot$;
    decided := FALSE;
    delivered := FALSE;
    phase := 0;
    vote := $\bot$;
    delivered_V := FALSE;
    collection_V := $\emptyset$;
    collection_B := $\emptyset$;
    noop := FALSE;
    phase0 := 0;

upon event <anbac, Propose | v> do
    decision := v;
    vote := v;
    if $i = 1$ then
        trigger <pl, Send | $P_2$, decision>;
    if $i = 1$ then
        set timer to time $n + 1$;
        phase := 2;
    else
        set timer to time $i$;
        phase := 1;
    if $v = 0$ then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, [V, 0]>;
        set timer0 to time 3;
    else
        set timer0 to time 2;

upon event <pl, Deliver | p, [V, 0]> do
    decision := 0;
    delivered_V := TRUE;
    trigger <pl, Send | p, [ACK, V]>;

upon event <pl, Deliver | p, [B, 0]> do
    decision := 0;
    trigger <pl, Send | p, [ACK, B]>;

upon event <timer0, Timeout> and vote = 1 and delivered_V and phase0 = 0 do
    forall $q \in \Omega$ do
        trigger <pl, Send | q, [B, 0]>;
    set timer0 to time 4;
    phase0 := 1;

upon event <pl, Deliver | p, [ACK, V]> do
    collection_V := collection_V $\cup \{p\}$;

upon event <pl, Deliver | p, [ACK, B]> do
    collection_B := collection_B $\cup \{p\}$;

upon event <timer0, Timeout> and vote = 0 do
    if collection_V = $\Omega$ and decided = FALSE then
        decided := TRUE;
        trigger <anbac, Decide | 0>;
    else
        noop := TRUE;

upon event <timer0, Timeout> and vote = 1 and delivered_V and phase0 = 1 do
    if collection_B = $\Omega$ and decided = FALSE then
        decided := TRUE;
        trigger <anbac, Decide | 0>;
    else
        noop := TRUE;

upon event <pl, Deliver | p, v> do
    decision := decision AND v;
    if phase $\leq 2$ then
        if $p = P_{(i-1)\%n}$ then
            delivered := TRUE;
    else if not decided then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, decision>;

upon event <timer, Timeout> and phase = 1 do
    if delivered = FALSE then
        decision := 0;
    if decision = 1 then
        trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
    else if $i = n$ then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, decision>;
    delivered := FALSE;
    if $i \geq f + 1$ then
        set timer to time $n + 2f + 1$;
        phase := 3;
    else
        set timer to time $n + i$;
        phase := 2;

upon event <timer, Timeout> and phase = 2 do
    if delivered = FALSE then
        decision := 0;
    if decision = 1 and $i \neq f$ then
        trigger <pl, Send | $P_{(i+1)\%n}$, decision>;
    if decision = 0 then
        forall $q \in \Omega$ do
            trigger <pl, Send | q, decision>;
    delivered := FALSE;
    set timer to time $n + 2f + 1$;
    phase := 3;

upon event <timer, Timeout> and phase = 3 and not decided do
    if decision = 1 and not noop then
        decided := TRUE;
        trigger <anbac, Decide | decision>;
```

Proof. (Proof of correctness of aNBAC.)
Termination. We only need to prove that a process decides in a failure-free execution. Clearly, a process proposes a vote before or at time 1. If every process proposes 1, then every process eventually timeouts at time $n + 2f + 1$, and *noop* is never assigned to TRUE. In a failure-free execution, every process has their local variable *delivered* to be TRUE at their timeout. As a result, at time $n + 2f + 1$, every process decides 1. Otherwise, if some process votes 0, then every process

who votes 0 timeouts at time 3 and decides while every process who votes 1 timeouts eventually at time 4 and also decides, Thus every process decides in a failure-free execution.

Commit-Validity. We only need to prove validity in every crash-failure execution. If process P decides 1, then P decides at time $n+2f+1$ when P's local variable *decision* $=1$ and *noop* is FALSE. Since *decision* $=1$, this leads to three facts: (a) that P has received no 0 from other processes; (b) that P's local variable *delivered* is TRUE when the timeout event for phases 1 or 2 occurs at P; and (c) that P proposes 1. If P is P_n, then according to (b), P has received 1 at phase 1, which means that the logical AND of all votes is 1 and every process proposes 1. If P is among $P_1, P_2, \ldots, P_f$, then according to (b), P has received 1 at phase 2, which implies that every process proposes 1. If P is not among $P_1, P_2, \ldots, P_f, P_n$, then according to (a), P does not receive 0 from $P_n, P_1, \ldots, P_f$ at time $n+1, \ldots, n+f+1$ respectively. Since at most f processes can crash, at least one process Q among $P_n, P_1, \ldots, P_f$ is instructed by the protocol to not send 0 to P. This implies that Q has received 1 at phase 2 if Q is among $P_1, P_2, \ldots, P_f$ or Q has received 1 at phase 1 if Q is P_n. Therefore, every process proposes 1.

Abort-Validity. We only need to prove validity in every crash-failure execution. If process P decides 0, then P only decides 0 at time 3 or at time 4. Then P either has voted 0, or has received a vote of 0 from some other process. Therefore, the abort-validity property is satisfied.

Agreement. By contradiction. Suppose that two different processes P and Q decide 1 and 0 respectively, in a network-failure execution. Then P decides at time $n+2f+1$ and Q decides at time 3 or at time 4.

When Q decides at time t ($t=3$ or 4), Q must have received an [ACK, V] or [ACK, B] from each process before or at t. On the other hand, when P decides, P's local variable *decision* is 1, which means that P has not received any message [B, 0] or [V, 0] before or when P decides. Since $n+2f+1 \geq 2+2+1=5>t$, P cannot manage to send [ACK, V] or [ACK, B] so that Q receives the message before or at t. A contradiction. □

E.4 (2n-2)NBAC

As in (n-1+f)NBAC, the timer here starts at time 1 when the first sending event happens.

```
Implements:
    (2n-2)MessageAtomicCommit, instance (2n-2)nbac.

Uses:
    PerfectPointToPointLinks, instance pl.
    Timer, instance timer.

upon event <(2n-2)nbac, Init> do
    votes := 1;
    received_B := FALSE;
    phase := 0;
    collection := {$P_i$};

upon event <(2n-2)nbac, Propose | v> do
    votes := votes AND v;
    if $1 \leq i \leq n-1$ then
        trigger <pl, Send | $P_n$, [V, v]>;
        set timer to time 3;
    else
        set timer to time 2;

upon event <pl, Deliver | p, [V, v]> do
    votes := votes AND v;
    collection := collection ∪ {p};

upon event <timer, Timeout> and phase = 0 and i = n do
    if votes = 1 and collection = Ω then
        forall q ∈ Ω do
            trigger <pl, Send | q, [B, 1]>;
    else
        votes := 0;
        forall q ∈ Ω do
            trigger <pl, Send | q, [B, 0]>;
    set timer to time $3+f$;
    phase := 1;

upon event <timer, Timeout> and phase = 0 and $1 \leq i \leq n-1$ do
    if received_B = FALSE then
        forall q ∈ Ω do
            trigger <pl, Send | q, [B, 0]>;
        votes := 0;
    set timer to time $3+f$;
    phase := 1;

upon event <pl, Deliver | p, [B, v]> do
    received_B := TRUE;
    votes := v;
    if votes = 0 then
        forall q ∈ Ω do
            trigger <pl, Send | q, [B, 0]>;

upon event <timer, Timeout> and phase = 1 do
    trigger <(2n-2)nbac, Decide | votes>;
```

Proof. (Proof of correctness of (2n-2)NBAC.) We show that every crash-failure execution of (2n-2)NBAC solves NBAC. While doing so, we show that every execution of (2n-2)NBAC satisfies validity and termination. Recall that in every crash-failure execution, every message arrives in time while in an execution, timeouts may be violated.

Termination. Every correct process decides at time $3+f$.

Commit-Validity. In every execution, if a process P decides 1, then at time $3+f$, the local variable *votes* is

1. If $P = P_n$, then at time 2, P must have received all n votes which are all 1. If $P \neq P_n$, then at time 3, P must have received a message [B, 1] from P_n, which implies that all processes vote 1.

Abort-Validity. In every execution, if a process P decides 0, then at time $3+f$, the local variable *votes* is 0. If $P = P_n$, then P votes 0, or receives a vote of 0 at time 2, does not receive some vote at time 2 or receives a message of [B, 0] (but sends a message of [B, 1] at time 2). The last two imply the crash of some process or the delay of some message. If $P \neq P_n$, then P votes 0, receives a message of [B, 0] from P_n at time 3, or does not receive any message from P_n at time 3, or receives a message of [B, 0] from some process (but receives a message of [B, 1] from P_n at time 3). The last two imply the crash of P_n or the delay of some message from P_n. Therefore, in every execution, if a process decides 0, then some process proposes 0 or a failure occurs.

Agreement. By contradiction. Suppose that E is a crash-failure execution such that two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. Then by the *commit-validity* property, every process votes 1. If $P = P_n$, then P has received all votes from all processes; since P decides at time $3+f$, P manages to send [B, 1] to every process and then Q should decide 1, which leads to a contradiction. If $P \neq P_n$, then P does not receive any message [B, 0] and moreover, P receives [B, 1] at time 3 from P_n. Clearly, For Q to decide 0, P_n must have crashed while sending [B,1] at time 2. (Otherwise, all have received [B,1], none sends message [B, 0] and then all decide 1. A contradiction.) Now that P_n crashes, Q must have received message [B, 0] later than time $2+f$. (Otherwise, P would receive a message of [B, 0] from Q earlier than or at time $3+f$ and thus decides 0, which is a contradiction.) Then between time 2 and time $f+2$, at least $f+1$ processes manage to send message [B,0] to some process and then crash, which contradicts the fact that at most f processes may crash. □

E.5 avNBAC

As (n-1+f)NBAC, the timer here starts at time 1 when the first sending event happens.

```
Implements:
    AVMessageAtomicCommit, instance avnbac.

Uses:
    PerfectPointToPointLinks, instance pl.
    Timer, instance timer.

upon event <avnbac, Init> do
    votes := 1;
    received_B := FALSE;
    collection := {$P_i$};

upon event <avnbac, Propose | v> do
    votes := votes AND v;
    if 1 $\le$ i $\le$ $n-1$ then
        trigger <pl, Send | $P_n$, [V, v]>;
        set timer to time 3;
    else
        set timer to time 2;

upon event <pl, Deliver | p, [V, v]> do
    votes := votes AND v;
    collection := collection $\cup$ {p};

upon event <timer, Timeout> and phase = 0 and i = n do
    if collection = $\Omega$ then
        forall q $\in$ $\Omega$ do
            trigger <pl, Send | q, [B, votes]>;
        trigger <avnbac, Decide | votes>;

upon event <timer, Timeout> and phase = 0 and 1 $\le$ i $\le$ $n-1$ do
    if received_B = TRUE then
        trigger <avnbac, Decide | votes>;

upon event <pl, Deliver | p, [B, v]> do
    received_B := TRUE;
    votes := v;
```

Proof. (Proof of correctness of avNBAC.)
Termination. For every correct process P, P proposes a vote and sets a timer. Then in a failure-free execution, P_n delivers every vote from every other process before or at time 2 and decides at time 2, while $P_i, i \in \{1, 2, \ldots, n-1\}$ delivers [B, b] before or at time 3 and decides at time 3.

Commit-Validity. If a process P decides 1, then P_n must have received all n votes which are all 1. I.e., every process proposes 1.

Abort-Validity. If a process P decides 0, then P_n must have received all n votes among which at least one is 0. I.e., some process proposes 0.

Agreement. By contradiction. Suppose that E is a network-failure execution such that two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. However, since both P and Q should decide the logical AND of all n votes, thus P and Q cannot decide differently. A contradiction. □

E.6 (2n-2+f)NBAC

We describe our (2n-2+f)NBAC protocol. Here if $f-1 = n$, then the condition $f-1 \le i \le n-1$ is never fulfilled no matter what i is (and thus the related events are never triggered). As (n-1+f)NBAC, we also change the timer slightly from the other protocols: the timer here starts at time 1 when the first sending event happens.

Implements:

```
  (2n-2+f)MessageAtomicCommit, instance (2n-2+f)nbac.

Uses:
  PerfectPointToPointLinks, instance pl.
  Timer, instance timer.
  UniformConsensus, instance uc.

upon event <(2n-2+f)nbac, Init> do
  votes := 1;
  received_V := FALSE;
  received_B := FALSE;
  received_Z := FALSE;
  phase := 0;
  decided := FALSE;
  proposed := FALSE;

upon event <(2n-2+f)nbac, Propose | v> do
  votes := votes AND v;
  if $i = 1$ then
    trigger <pl, Send | $P_2$, [V, v]>;
    set timer to time $n + 1$;
    phase := 1;
  else
    set timer to time $i$;

upon event <pl, Deliver | p, [V, v]> and $phase = 0$ do
  votes := votes AND v;
  received_V := TRUE;

upon event <timer, Timeout> and $phase = 0$ do
  if received_V = TRUE then
    if $i = n$ then
      trigger <pl, Send | $P_1$, [B, votes]>;
    else
      trigger <pl, Send | $P_{i+1}$, [V, votes]>;
  else
    votes := 0;
    if proposed = FALSE then
      trigger <uc, Propose | 0>;
      proposed := TRUE;
  set timer to time $n + i$;
  phase := 1;

upon event <pl, Deliver | p, [B, b]> and $phase = 1$ do
  votes := votes AND b;
  received_B := TRUE;

upon event <timer, Timeout> and $phase = 1$ and $i =$
f do
  if received_B = TRUE then
    trigger <pl, Send | $P_{f+1}$, [B, votes]>;
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
  else
    votes := 0;
    if proposed = FALSE then
      trigger <uc, Propose | 0>;
      proposed := TRUE;
  phase = 2;

upon event <timer, Timeout> and $phase = 1$ and $i =$
n do
  if received_B = TRUE then
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
    if $f \geq 2$ then
      trigger <pl, Send | $P_1$, [Z, votes]>;
  else
    if proposed = FALSE then
      trigger <uc, Propose | votes>;
      proposed := TRUE;

upon event <timer, Timeout> and $phase = 1$ and $1 \leq$
i $\leq f - 1$ do
  if received_B = TRUE then
    trigger <pl, Send | $P_{i+1}$, [B, votes]>;
  else
    votes := 0;
    if proposed = FALSE then
      trigger <uc, Propose | 0>;
      proposed := TRUE;
  set timer to time $2n + i$;
  phase := 2;

upon event <timer, Timeout> and $phase = 1$ and $f +$
$1 \leq i \leq n - 1$ do
  if received_B = TRUE then
    trigger <pl, Send | $P_{i+1}$, [B, votes]>;
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
  else
    forall $q \in \{P_1, P_2, \ldots, P_f, P_n\}$ do
      trigger <pl, Send | q, [HELP]>;

upon event <pl, Deliver | p, [HELP]> and $i = n$ and
$phase = 1$ do
  trigger <pl, Send | p, [HELPED, votes]>;

upon event <pl, Deliver | p, [HELP]> and $1 \leq i \leq f$
and $phase = 2$ do
  trigger <pl, Send | p, [HELPED, votes]>;

upon event <pl, Deliver | p, [HELPED, v]> and not proposed do
  trigger <uc, Propose | v>;
  proposed := TRUE;

upon event <pl, Deliver | p, [Z, z]> and $phase = 2$ do
  votes := votes AND z;
  received_Z := TRUE;

upon event <timer, Timeout> and $phase = 2$ and $1 \leq$
i $\leq f - 1$ do
  if received_Z = TRUE then
    if decided = FALSE then
      trigger <(2n-2+f)nbac, Decide | votes>;
      decided := TRUE;
```

```
        if $f - 1 \geq i + 1$ then
            trigger <pl, Send | $P_{i+1}$, [Z, votes]>;
    else
        if proposed = FALSE then
            trigger <uc, Propose | votes>;
            proposed := TRUE;

upon event <uc, Decide | d> and not decided do
    trigger <(2n-2+f)nbac, Decide | d>;
    decided := TRUE;
```

Proof. (Proof of correctness of (2n-2+f)NBAC.)
Termination. Consider any crash-failure (or network-failure) execution E. In E, every correct process proposes a vote. For a correct process $P_i, i \in \{f, n\}$, the timer eventually timeouts at time $n + i$; at time $n + i$, P_i either decides without invoking *uc* in (2n-2+f)NBAC or proposes a value to *uc*. For a correct process $P_i, i \in \{1, 2, \ldots, f - 1\}$, P_i eventually timeouts at time $2n + i$; at time $2n + i$, P_i decides without invoking *uc* or proposes a value to *uc*. For a correct process $P_i, i \in \{f + 1, f + 2 \ldots, n - 1\}$, P_i eventually timeouts at time $n + i$; at time $n + i$, P_i either decides at time $n + i$ or queries $P_1, P_2, \ldots, P_f, P_n$ for help. If P_n is correct, then P_n eventually assigns 1 to *phase*; if a process in $\{P_1, P_2, \ldots, P_f\}$ is correct, then the process eventually assigns 2 to *phase*. Since at most f processes may crash, then at least one process in $\{P_1, P_2, \ldots, P_f, P_n\}$ is correct and therefore P_i receives at least one message [HELPED, *] and then proposes a value to *uc*. Thus by the termination property of *uc* in a crash-failure system (or in a network-failure system), every correct process decides in E.

Commit-Validity. In every execution, if a process P decides 1, then P's decision is either a decision of *uc* or not. If P's decision is not a decision of *uc*, then P decides its local variable *votes* = 1 and there are four possibilities for P when P decides: (1) $P = P_f$, *phase* = 1, *received_B* is TRUE; (2) $P = P_n$, *phase* = 1, *received_B* is TRUE; (3) $P \in \{P_{f+1}, P_{f+2}, \ldots, P_{n-1}\}$, *phase* = 1, *received_B* is TRUE; (4) $P \in \{P_1, P_2, \ldots, P_{f-1}\}$, *phase* = 2, *received_Z* is TRUE. By the protocol, in each of the four possibilities, *votes* is the logical AND of all n votes and thus every process proposes 1. If P's decision is a decision of *uc*, then some process Q proposes *votes* = 1 or $v = 1$ to *uc* and therefore there are three possibilities when Q proposes: (1) $Q = P_n$, *phase* = 1, *received_B* is FALSE, *received_V* is TRUE and Q proposes *votes*; (2) $Q \in \{P_{f+1}, P_{f+2}, \ldots, P_{n-1}\}$, *phase* = 1, *received_B* is FALSE, Q delivers message [HELPED, v] from some process $p \in \{P_1, P_2, \ldots, P_f, P_n\}$ and Q proposes v; (3) $Q \in \{P_1, P_2, \ldots, P_{f-1}\}$, *phase* = 2, *received_Z* is FALSE, *received_B* is TRUE, *received_V* is TRUE, and Q proposes *votes*. By the protocol, in each of the three possibilities, *votes* or v is the logical AND of all n votes and thus every process proposes 1.

Abort-Validity. In every execution, if a process P decides 0, then P's decision is either a decision of *uc* or not. If P's decision is not a decision of *uc*, then P decides its local variable *votes* = 0; since *votes* is the logical AND of all n votes and thus some process proposes 0. If P's decision is a decision of *uc*, then some process Q proposes 0 to *uc*. If Q proposes Q's local variable *votes*, then again *votes* is the logical AND of all n votes and thus some process proposes 0. If Q proposes v where Q delivers message [HELPED, v] from some process $p \in \{P_1, P_2, \ldots, P_f, P_n\}$, then there are two possibilities when Q proposes to *uc*: (1) $p = P_n$, *phase* = 1, *received_V* is FALSE; (2) $p \in \{P_1, P_2, \ldots, P_f\}$, *phase* =2, *received_B* is FALSE. We note that by the protocol, in every crash-failure execution where no process crashes, neither (1) nor (2) occurs. As a result, if (1) or (2) occurs, then some process must have crashed or some message must have been delayed. If Q proposes 0 to *uc*, then there are three possibilities when Q proposes 0 to *uc*: (1) $Q \in \{P_2, P_3, \ldots, P_n\}$, *phase* = 0, *received_V* is FALSE; (2) $Q = P_f$, *phase* = 1, *received_B* is FALSE; (3) $Q \in \{P_1, P_2, \ldots, P_{f-1}\}$, *phase* = 1, *received_B* is FALSE. By the protocol, in every crash-failure execution where no process crashes, none of the three possibilities occurs. As a result, if (1) or (2) occurs, then some process must have crashed or some message must have been delayed.

Agreement. By contradiction. Suppose that E is an execution such that two processes P and Q decide differently. W.l.o.g., P decides 1 and Q decides 0. Then by the *agreement* property of uniform consensus, at least one of P and Q's decisions is not a decision of *uc*. If P's decision is a decision of *uc*, then Q's decision is not a decision of *uc*; however, by the proof of the *commit-validity* property above, every process proposes 1 and thus when Q decides, Q's local variable *votes* = 1, which leads to a contradiction. If P's decision is not a decision of *uc*, then by the proof of the *commit-validity* property above, every process proposes 1 and moreover, Q's decision must be a decision of *uc*; as a result, some process R proposes 0 or v to *uc*. When P decides, if a process in $\{P_1, P_2, \ldots, P_f\}$ has not yet crashed, then its local variables *phase* = 2, *received_B* is TRUE (when *phase* is assigned to 2), *received_V* is TRUE (when *phase* is assigned to 2); if a process in $\{P_{f+1}, P_{f+2}, \ldots, P_n\}$ has not yet crashed, then its local variables *phase* = 1, *received_V* is TRUE (when *phase* is assigned to 1). Therefore, R cannot propose 0 when at R, *phase* = 0 or 1; since *received_V* and *received_B* can only be assigned to TRUE by the protocol (except for initialization), no process would send message [HELPED, 0] to R and thus R cannot propose $v = 0$. As a result, R does not exist, which gives rise to a contradiction.

□