The Two-Phase Commitment Protocol in an Extended π-Calculus

Martin Berger

Department of Computing
Imperial College, London
Email: M.Berger@doc.ic.ac.uk

Kohei Honda

Department of Computer Science
Queen Mary and Westfield College, London
Email: kohei@dcs.qmw.ac.uk

Abstract

We examine extensions to the π-calculus for representing basic elements of distributed systems. In spite of its expressiveness for encoding various programming constructs, some of the phenomena inherent in distributed systems are hard to model in the π-calculus. We consider message loss, sites, timers, site failure and persistence as extensions to the calculus and examine their descriptive power, taking the *Two Phase Commit Protocol* (2PCP), a basic instance of an atomic commitment protocol, as a testbed. Our extensions enable us to represent the 2PCP under various failure assumptions, as well as to reason about the essential properties of the protocol.

1 Introduction

The field of process calculi has seen major advances in the decades since the introduction of CCS [21], CSP [14] and ACP [5]. In particular, with the advent of the π-calculus [23] and other name passing process calculi [9,15,24,26], it has been found that diverse computational structures in both sequential and concurrent computing are uniformly representable as interacting processes. This enables us to apply standard syntactic reasoning methods developed for process calculi to a wide variety of computational phenomena. However, in spite of its high expressive power and its interaction-based computing model, the π-calculus as presently given does not suffice for sound description of basic elements of distributed computing systems. This is because some operations and phenomena which frequently arise in distributed systems are difficult to

This is a preliminary version. The final version can be accessed at
URL: http://www.elsevier.nl/locate/entcs/volume39.html

decompose into name passing: they are often too "low-level," in the sense that they represent computational mechanisms left implicit or not treated in the π-calculus, loss of message in transit, timers, manipulation of process activities, process failure and recovery being examples. Some of these are such that their satisfactory (say compositional and fully abstract) encodings cannot exist: for others, even if a translation into name passing would be possible, the description using such translations suffers basic problems such as lack of compositionality, lack of extensibility or excessive complexity, making reasoning cumbersome, if not impossible. Thus extensions cleanly representing these phenomena are needed at least for convenience in the sense that they should aid modelling distributed systems in a way faithful to real computing phenomena, as well as offering useful reasoning frameworks. In this context one may observe that, as far as the described phenomena are representable by known computing devices, they are in some way encodable into Turing machines. However this does not mean Turing machines provide a convenient framework for modelling distributed computing systems!

But what kinds of extensions to the π-calculus should we consider for modelling distributed systems? We wish them to be *basic* and *incremental*, in the sense that combinations of a few simple extensions can represent a wide range of phenomena, essential to distributed systems; and that these constructs interact with each other consistently, so their incremental addition leads to a feasible increase in the complexity of the equational frameworks. If well-chosen, such constructs should be equipped with clean operational semantics, both in terms of pure dynamics (reduction) and behavioural semantics (labelled transitions). The latter is important since equational reasoning based on labelled transition offers a convenient method for exploring semantic properties without having to resort to quantification over all possible contexts. More generally, the constructs should be able to represent distributed systems with clarity and rigour so that descriptions become amenable to formal analysis as well as intuitive understanding to help putting the study of distributed systems on a uniform technical footing to aid comparison, integration and further development.

Against this background, the present paper studies a few extensions to an asynchronous version of the π-calculus for representing distributed computation, and examines their descriptive power for the description and correctness proof of an important distributed algorithm, the *Two Phase Commit Protocol* (hereafter 2PCP). The extended constructs are chosen to cover, in as simple as possible a way, what we regard as some of the essential phenomena of distributed systems, namely message loss, timers, process failure and savepoints. As they are omnipresent in distributed computing, these phenomena must, in some way or other, be captured when representing distributed systems. While we are far from claiming our extensions are comprehensive enough, they together offer a coherent framework for describing and reasoning about extended processes based on the labelled transitions. The description and the correctness proof of the 2PCP using the extended constructs demonstrate their expressive power and applicability. As the 2PCP deals with key elements of distributed computing systems including message loss, timers and process failures/recoveries, it is an ideal testbed for our purpose. A clean description of the 2PCP is obtained under various failure assumptions, and the correctness proof of its central property is obtained using the equations inherent in added constructs.

Since the present paper uses the 2PCP not only as an application but also for motivating our extended constructs, general illustration of this protocol would be due. The 2PCP is the most basic instance of an *atomic commitment protocol* [6,11,12,20], which achieves basic properties of transactions, notably atomicity, in the presence of (partial) failures in distributed systems. By atomicity we mean that a transaction is committed if and only if all the transactions it depends on also commit. The 2PCP achieves atomic commitment under the assumption that all occurring failures are of the following two kinds.

- Messages either get delivered accurately or disappear without a trace ("no forging").
- Processes either work correctly or they fail by stopping completely. As long as a process is stopped, it does not engage in any interaction or state change ("fail-stop" [27]). Stopped processes may or may not restart later.

In particular, it assumes the absence of Byzantine behaviour [19]. Under these assumptions, the 2PCP is known to achieve atomic commitment, using timers and savepoints as countermeasures against the failures noted above. Based on the 2PCP, other atomic commitment protocols have been developed, which are more efficient with respect to certain metrics (such as the number of messages to be sent to achieve commitment, or the likelihood of blocking) [6,12,28,29]. While being most basic among atomic commitment protocols, the 2PCP has fairly complex behaviour, due to the possibilities of failures and the incorporation of mechanisms to cope with them. We are not aware of any previous work offering a fully formal description of this algorithm. The extensions to the π-calculus we introduce in the present paper are simple, but they are powerful enough to concisely and cleanly represent, as well as reason about the full 2PCP. The high-level structure of both the description of the protocol and the correctness of the proofs remain stable for all versions of the protocol under different failure assumptions (most of which are presented only in the full version of the present paper [4]). This may be seen as good evidence that our extensions coherently capture basic elements of distributed computing. In comparison with the description of the 2PCP in representative textbooks on transaction processing [6,12], the present approach differs in that it captures the whole of the interactive behaviour of the protocol in a compositional way: all participants of the 2PCP are described as interacting processes, and their composition formally defines what the behaviour of the 2PCP is in a mathematically unambiguous way. This enables us to use behavioural semantics for formulating and reasoning about atomicity, a fundamental property of the protocol. Another payoff of our formalisation was that we identified two subtle problems in the classic presentation of the 2PCP.

This paper is a technical summary of [4] to which readers may refer for full proofs and further details. In the remainder, Section 2 gives an informal description of the 2PCP. Section 3 presents the base calculus, and shows how it can be used for representing the core protocol without the assumption of failures. Section 4 studies our few extensions to the base calculus, message loss, timers, process failure and savepoints, and uses them to represent the 2PCP. Section 5 outlines how the description in Section 4 enables us to establish the central property of the 2PCP, concentrating on key technical ideas and the use of algebraic laws. Section 6 is devoted to discussions, including observations on relative expressiveness of the added constructs with respect to the original calculus.

2 Behaviour of the 2PCP: an Informal Description

The 2PCP [6,11,12,20] is a *distributed* protocol, in the sense that it consists of multiple possibly faulty processes that interact via possibly faulty channels. It has one transaction manager, which we hereafter call *coordinator*, and participating (sub)transaction, which we call *participants*. The principal objective of this protocol is to ensure that, as far as outside observers can tell, all participating transactions commit together (usually writing some datum to a persistent storage, though they can include other actions) or abort together. This is the *atomicity property* of the protocol. Below we outline the basic behaviour of 2PCP. First we describe the central part of the protocol, which we call the *core protocol*, assuming the absence of failures.

(1) Each participant sends to the coordinator a message containing its vote (abort or commit). If its message is abort, the participant will itself abort immediately. If not, it waits for the votes from all the participants. The coordinator waits for these messages. The coordinator itself can decide to commit or abort.

(2) When all the participants as well as the coordinator have voted to commit, the coordinator will send all participants a message telling them to commit. A participant which has voted to commit would now commit, once it receives this message.

(3) If there is any abort vote (including the vote by the coordinator), the coordinator sends all the participating transactions a message ordering them to abort. A participant which has voted to commit would now abort on receiving this message.

Since the protocol should work in distributed environments, messages can be lost in transit. To cope with it, the 2PCP uses a *timer*. Using timers, the above core protocol is augmented as follows.

(T1) In (1), the coordinator sets a timer before starting to wait for votes: if the timer expires, it decides to abort.

(T2) Similarly, in (1), a participant who has voted to commit, waits for a decision after setting up its own timer: if the timer expires, it assumes the decision message is lost, and requests the coordinator to give the decision again, after setting up another timer (the request can again be lost, in which case the participant's timer expires so the same procedure is repeated) [1].

Another type of failure is *process failure*, i.e. the possibility a system can crash. We assume all crashed systems will eventually restart, cf. [6,12]. It is crucial that processes restart in a consistent manner after a process failure: To this end, the protocol is further augmented by *savepoints* [7]. Roughly, a savepoint S of a process P is a process as a persistent datum such that if P recovers from a crash, it will be reincarnated as S. We augment the protocol as follows:

(S1) The initial savepoint of the coordinator is a process such that, after restart, it will order participants to abort. This is because a crashed coordinator is regarded as untrustworthy.

(S2) After the transaction manager has received all the votes from the participants and all are for commit, the coordinator makes a savepoint that will order "commit" to all participants. This savepoint should be made persistent *before* any orders are sent out.

(S3) For participants, the initial savepoint is a process that aborts, while, after it voted to commit, its savepoint is such that, when restarted, again to vote to commit and to wait for the decision. This savepoint should again be made persistent *before* the vote is sent.

The essence of the 2PCP as a distributed protocol lies in that the core protocol is capable of harnessing these additional mechanisms so that its key properties such as atomicity are maintained in the presence of failures.

3 Representing the 2PCP in the π-Calculus (1): The Core Protocol

This section presents the base calculus used in our present study. We use the calculus to represent the core part of the 2PCP, and discusses a basic semantic property of the protocol we can state for the description.

[1] A variation of the protocol would obtain this information from other participants rather than from the coordinator. Such 2PCPs are called *decentral*. They are advantageous in that the coordinator is not a single point of failure in the last phase of the protocol. To keep proofs simple, we shall not deal with decentral 2PCPs in this text even though an adaptation would simple.

3.1 A Basic π-Calculus

As in many distributed protocols, the 2PCP is based on asynchrony in messages. Further, information flow with respect to binary decision (abort or commit) plays a central role in its description and analysis. For describing these features, we choose the asynchronous version of the π-calculus [16] augmented with branching [30]. The role of branching in semantic arguments will become clear later. Let $a, b, c, \ldots, x, y, z, \ldots$ range over *names*. The syntax of *processes*, written $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \ldots$, is given by the following grammar.

$$\begin{aligned}\mathsf{P} ::= &\ x(\vec{y}).\mathsf{P} \mid \overline{x}\langle\vec{y}\rangle \mid \mathsf{P}|\mathsf{Q} \mid (\nu x)\mathsf{P} \mid \mathsf{0}\\ &\mid\ !x(\vec{y}).\mathsf{P} \mid !\overline{x}\langle\vec{y}\rangle \mid x[(\vec{y}).\mathsf{P}, (\vec{z}).\mathsf{Q}] \mid \overline{x}\mathsf{left}\langle\vec{y}\rangle \mid \overline{x}\mathsf{right}\langle\vec{y}\rangle\end{aligned}$$

The last three constructs are called *branching input* and (*left* and *right*) *branching output*, and perform branching at the time of interaction. This construct is easily encodable in the calculus without these constructs: however, in the presence of distributed failures, the known simple encoding does not work, see Section 6. They also play an essential role in equational reasoning. The notion of free and and bound names, written fn(P) and bn(P), as well as α-convertibility $\equiv_\alpha$ are standard. Throughout the paper we assume the natural sorting discipline [30]. When no name passing is used, we write $x.P$ for input, $\overline{x}$ for output, $x[\mathsf{P}, \mathsf{Q}]$ for a branching input, and $\overline{x}\mathsf{left}$ and $\overline{x}\mathsf{right}$ for branching outputs. Leaving the standard definition of structural rules and reduction to [4], we here only record the dynamics of branching (without name passing): $x[\mathsf{P}, \mathsf{Q}] \,|\, \overline{x}\mathsf{left} \longrightarrow \mathsf{P}$, and $x[\mathsf{P}, \mathsf{Q}] \,|\, \overline{x}\mathsf{right} \longrightarrow \mathsf{Q}$. The corresponding labelled transitions $\xrightarrow{\pi}$ are easily obtained, using $\overline{x}\mathsf{left}\langle(\nu\vec{y})\vec{z}\rangle$ and $x\mathsf{left}((\nu\vec{y})\vec{z})$ (and the symmetrically for the right branch) as labels. The standard strong and weak bisimilarities (the latter subsequenty often referred to simply as bisimilarity) are denoted $\sim$ and $\approx$. We also use the notation $\mathsf{P} \oplus \mathsf{Q}$, the internal sum of P and Q, which stands for $(\nu c)(c.P|c.Q|\overline{c})$ with c fresh. Clearly $\mathsf{P} \oplus \mathsf{Q} \longrightarrow \mathsf{P}' \sim \mathsf{P}$ and $\mathsf{P} \oplus \mathsf{Q} \longrightarrow \mathsf{Q}' \sim \mathsf{Q}$, which are the only one-step reduction $\mathsf{P} \oplus \mathsf{Q}$ owns. Please note that the expressive power of the π-calculus is *not* needed to model the 2PCP (assuming no failures).

3.2 Description of the 2PCP (1): the Core Protocol

If we assume the absence of failure, i.e. if we only deal with the core protocol in the sense of Section 2, then the 2PCP can be described using the basic calculus just introduced. The representation is simple and serves as a basic reference point on which further constructs would be built. For readability, we use symbols such as vote_i for channels to describe the meaning of the messages they would carry (for example vote_i is used as the channel for a vote by the i-th participant). One of the basic aspects of the representation is that it gives the behaviour of the protocol as seen by external observers. The representation is denoted 2PCP, and is given by the following configuration.

$$2\mathsf{PCP} = (\nu \mathsf{vote}_{self})(\nu \vec{\mathsf{vote}})(\nu \vec{\mathsf{dec}})(\mathsf{C} \mid \mathsf{P}_1 \mid \ldots \mid \mathsf{P}_n).$$

The protocol is the composition of one coordinator and n participants. Channels used for communication among a coordinator and participants are hidden. The coordinator C is again a composition of several subprocesses:

$$\mathsf{C} = (\nu \vec{c})(\nu c_{self})(\nu a)(\mathsf{C}_{wait} \mid \mathsf{C}^{true}_{end} \mid \mathsf{C}^{false}_{end} \mid \mathsf{C}_{self})$$

where C_{wait} is a process which waits for votes from other processes, including its own one. C^{true}_{end} is a process which, when the votes are unanimously "commit", would send out the "commit" decision to participants. C^{false}_{end}, on the other hand, would send out the "abort" decision if any one of participants (or itself) sends an abort vote. Finally C_{self} is a process which nondeterministically decides whether it wants to abort or to commit.

$$\begin{aligned}
\mathsf{C}_{wait} &= \mathsf{vote}_1[\overline{c_1}, \overline{a}] \mid \ldots \mid \mathsf{vote}_n[\overline{c_n}, \overline{a}] \mid \mathsf{vote}_{self}[\overline{c_{self}}, \overline{a}] \\
\mathsf{C}_{self} &= \overline{\mathsf{vote}_{self}}\mathsf{left} \ \oplus \ \overline{\mathsf{vote}_{self}}\mathsf{right} \\
\mathsf{C}^{true}_{end} &= c_1. \ldots .c_n.c_{self}.(\overline{\mathsf{dec}_1}\mathsf{left} \mid \ldots \mid \overline{\mathsf{dec}_n}\mathsf{left}) \\
\mathsf{C}^{false}_{end} &= a.(\overline{\mathsf{dec}_1}\mathsf{right} \mid \ldots \mid \overline{\mathsf{dec}_n}\mathsf{right})
\end{aligned}$$

Note that C^{true}_{end} needs $n+1$ commit votes to decide to commit, while C^{false}_{end} needs only one abort vote to decide to abort. We now give the representation of a participant, where P_i denotes the i-th participant.

$$\begin{aligned}
\mathsf{P}_i &= \mathsf{P}^{true}_i \oplus \mathsf{P}^{false}_i \\
\mathsf{P}^{true}_i &= \overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{dec}_i[!\overline{\mathsf{commit}_i}, \ !\overline{\mathsf{abort}_i}] \\
\mathsf{P}^{false}_i &= \overline{\mathsf{vote}_i}\mathsf{right} \mid !\overline{\mathsf{abort}_i}
\end{aligned}$$

To model two possible outcomes of voting in a participant, each participant consists of two branches of an internal sum, one voting to commit (and to wait for a decision from the coordinator) and the other voting to abort. The actions of committing and aborting are represented by outputting at special ports: in practice, they would contain various behaviours including writing to databases. Replication is not necessary, but simplifies reasoning. Using bisimilarity, we state a central property of the core protocol. It shows how a central property of atomic commitment protocols is cleanly translated into a statement on behavioural equivalence between processes. We do not prove the result here since it is subsumed by the equivalent result for the full protocol, which we discuss in Section 5.

Proposition 3.1 *Let* $\mathsf{Abort} = \Pi^n_{i=1}!\overline{\mathsf{abort}_i}$ *and* $\mathsf{Commit} = \Pi^n_{i=1}!\overline{\mathsf{commit}_i}$. *Then* $2\mathsf{PCP} \approx \mathsf{Abort} \oplus \mathsf{Commit}$.

4 Representing the 2PCP in the π-Calculus (2)

4.1 Extending the π-Calculus (1): Message Loss

In many distributed computing environments such as the Internet, a message can be lost during transmission (for example due to overflowing router buffers). The incorporation of message loss looks simple: just add the rule $\overline{x}\langle\vec{y}\rangle \longrightarrow \mathsf{0}$ (and mutatis mutandis for branching outputs). Alas this rule does not capture the reality of message loss: two processes in a shared memory multiprocessor computer are more realistically modelled without the possibility of message loss. One method is to have two kinds of channels, lossy (or non-dependable) ones and dependable ones, cf. [1]. However, in distributed systems, the same channel could be both reliable and unreliable depending on whether it is carrying a local message or a remote message. As an alternative way to realistically model message loss in distributed systems, we augment the calculus with "sites", and separate "internal message passing" (interaction within a site) from "external message passing" (interaction between two sites). The idea is that interaction within a site does not suffer from message loss, while any message travelling from its originating site to a remote site may disappear without a trace. To be able to determine whether a message is for communication within a site or for some other site, we impose a natural restriction on syntax of processes. Sites play an essential role in the subsequent development, not only for message loss but also for process failures and persistence.

The incorporation of sites is simple. Processes P are as defined before. Then *networks*, ranged over by $\mathsf{N}, \mathsf{N}', \ldots$, are given by the following syntax, where A is a finite set of names.

$$\mathsf{N} \quad ::= \quad [\mathsf{P}]_A \ \mid\ \mathsf{N}_1|\mathsf{N}_2 \ \mid\ (\nu x)\mathsf{N} \ \mid\ \mathsf{0}$$

Here $[\mathsf{P}]_A$ denotes a site which is ready to receive messages at names in A. We may consider $[\mathsf{P}]_A$ as a LAN connected to the Internet, in which case A may as well be the set of IP-addresses the hosts on the LAN own. Alternatively, we may consider $[\mathsf{P}]_A$ as a host, in which case A might be understood as containing all addresses of sockets that are serviced by P. Following networking community terminology, we call A the (set of) *access points* of $[\mathsf{P}]_A$. More generally, the set of *access points* of N, $\mathrm{ap}(\mathsf{N})$, is given by: $\mathrm{ap}([\mathsf{P}]_A) = A$, $\mathrm{ap}(\mathsf{N}_1|\mathsf{N}_2) = \mathrm{ap}(\mathsf{N}_1) \cup \mathrm{ap}(\mathsf{N}_2)$, $\mathrm{ap}((\nu x)\mathsf{N}) = \mathrm{ap}(\mathsf{N}) \setminus \{x\})$ and $\mathrm{ap}(\mathsf{0}) = \emptyset$. The overloaded operators $|$, (νx) and 0 are understood in the same way as the corresponding operators for processes and obey the same structural rules. The idea of using input interface for a process in a similar setting already appeared in the context of the join calculus [10]. We use the following well-formedness condition (here and henceforth we assume the standard variable convention for names). Given $\mathsf{Q} = x(y).\mathsf{P}$, we say x occurs in Q as an input subject and a free occurrence of y in P is *input-bound* by (y). We then say P is *local* if no input subject is input-bound. We say N is *well-formed*, written $\vdash \mathsf{N}$, if $\vdash \mathsf{N}$ is derivable using the following rules: (i) $\vdash 0$ is always derivable; (ii) $\vdash [\mathsf{P}]_A$ if

P is local and each free input subject in P is in A; and (iii) $\vdash \mathsf{N}_1|\mathsf{N}_2$ if $\vdash \mathsf{N}_1$ and $\vdash \mathsf{N}_2$ and, moreover, $\mathrm{ap}(\mathsf{N}_1) \cap \mathrm{ap}(\mathsf{N}_2) = \emptyset$; and (iv) $\vdash (\nu x)\mathsf{N}$ if $\vdash \mathsf{N}$. The free and bound names of processes and networks are entirely standard, but note that $\mathrm{fn}([\mathsf{P}]_A) = \mathrm{fn}(\mathsf{P}) \cup A$. From now on we assume all networks are well-formed. The structural congruence for processes, $\mathsf{P} \equiv \mathsf{Q}$, is just as in the basic calculus. The operators (νx), $|$ and $\mathsf{0}$ for networks obey the same structural rules as those for processes. In addition we set $[(\nu x)\mathsf{P}]_A \ \equiv\ (\nu x)[\mathsf{P}]_{A\cup\{x\}}$ [2]. Over networks $\equiv$ is the smallest congruence containing these rules. The reduction $\to$ over processes is unchanged, while that over networks, is given by the following rules.

$$(\textsc{Intra})\ \frac{\mathsf{P} \ \longrightarrow\ \mathsf{P}'}{[\mathsf{P}]_A \ \longrightarrow\ [\mathsf{P}']_A}$$

$$(\textsc{N-Com})\ [\ \mathsf{P}|x(\vec{y}).\mathsf{P}'\ ]_A \mid [\ \overline{x}\langle\vec{z}\rangle|\mathsf{Q}\ ]_B \ \longrightarrow\ [\ \mathsf{P}|\mathsf{P}'\{\vec{z}/\vec{y}\}\ ]_A \mid [\mathsf{Q}]_B$$

$$(\textsc{Loss})\ [\ \mathsf{P}|\overline{x}\langle\vec{z}\rangle\ ]_A \ \longrightarrow\ [\mathsf{P}]_A \qquad (x \notin A)$$

as well as the obvious rule for branching corresponding to (N-COM), and the standard rules which close $\longrightarrow$ under $|$, (νx) and $\equiv$, all assuming well-formedness. As an example of the use of (LOSS) we obtain the reduction of form $[\overline{x}\langle\vec{z}\rangle]_A \ \longrightarrow\ [\mathsf{0}]_A$ whenever $x \notin A$.

The corresponding labelled transitions are also concise. The rules for processes are identical to those in the basic calculus. For networks, we define transitions by the following rules.

$$(\textsc{Intra})\ \frac{\mathsf{P} \xrightarrow{\tau} \mathsf{Q}}{[\mathsf{P}]_A \xrightarrow{\tau} [\mathsf{Q}]_A}$$

$$(\textsc{N-Out})\ \frac{\mathsf{P} \xrightarrow{\overline{x}\langle(\nu\vec{y})\vec{z}\rangle} \mathsf{Q} \quad x \notin A \quad \{\vec{y}\} \cap A = \emptyset}{[\mathsf{P}]_A \xrightarrow{\overline{x}\langle(\nu\vec{y})\vec{z}\rangle} [\mathsf{Q}]_{A\cup\{\vec{y}\}}}$$

$$(\textsc{N-In})\ \frac{\mathsf{P} \xrightarrow{x((\nu\vec{y})\vec{z})} \mathsf{Q} \quad \{\vec{y}\} \cap A = \emptyset}{[\mathsf{P}]_A \xrightarrow{x((\nu\vec{y})\vec{z})} [\mathsf{Q}]_A}$$

$$(\textsc{Loss})\ \frac{\mathsf{P} \xrightarrow{\overline{x}\langle(\nu\vec{y})\vec{z}\rangle} \mathsf{Q} \quad x \notin A}{[\mathsf{P}]_A \xrightarrow{\tau} [(\nu\vec{y})\mathsf{Q}]_A}$$

There are also the obvious branching versions of (N-OUT), (N-IN) and the rules corresponding to the rules of the basic calculus. We also need a version of (ALPHA) for networks. As can be seen, access points play a central role for interaction between two networks. Detailed illustration of transition rules is relegated to [4]. Weak bisimilarity $\approx$ over networks is defined in the standard

[2] We do not add the rule $[\mathsf{0}]_A \ \equiv\ \mathsf{0}$ because, if we did, $\mathsf{M} \equiv \mathsf{N}$ would not imply $\mathrm{fn}(\mathsf{M}) = \mathrm{fn}(\mathsf{N})$ as $\mathrm{fn}([\mathsf{0}]_{\{x\}}) = \{x\} \neq \emptyset = \mathrm{fn}(\mathsf{0})$. A network $[\mathsf{0}]_A$ acts as a 'domain squatter' that cannot perform any computation but prevents other networks from utilizing names in A.

way. For well-formed networks, $\approx$ is a congruence with respect to all operators.

4.2 Extending the π-Calculus (2): Timers

If we can lose messages, the core protocol loses atomicity: it is now possible that the whole configuration deadlocks by the loss of, say, a vote. The 2PCP, as many other distributed algorithms, addresses this issue by using *timers.* Timers play a fundamental role in all areas of distributed software, including for example TCP, one of the core Internet protocols. The timer we choose in the present paper reflects the basic character of timers in distributed systems in a simple form. While there have been various attempts to incorporate the notion of (real-)time behaviour into process calculi, cf. [13], the addition of a timer to the π-calculus has been lacking so far. The syntax of processes is extended as follows.

$$\mathsf{P} \quad ::= \quad \ldots \mid \mathsf{timer}^t(\mathsf{Q},\mathsf{R})$$

where Q should be *input guarded*, i.e. prefixed by either a standard or a branching input. The *subject* of a timer $\mathsf{timer}^t(\mathsf{P},\mathsf{Q})$ is the initial input subject of P. t ranges over integers *greater* than 0. In $\mathsf{timer}^t(\mathsf{P},\mathsf{Q})$, t represents the clock ticks left before timeout of the timer. P and Q are the *timeout* and *timein continuation* of the timer. When it times out, the timer becomes Q. As long as it has not timed out, interaction with the input guard of P is possible: on having this input, the timer becomes the residual of P. The technical development of timers hinges on the following *time stepper function* ϕ.

$$\phi(\mathsf{P}) = \begin{cases} \mathsf{timer}^{t-1}(\mathsf{Q},\mathsf{R}) & \mathsf{P} = \mathsf{timer}^t(\mathsf{Q},\mathsf{R}), t > 1 \\ \mathsf{R} & \mathsf{P} = \mathsf{timer}^t(\mathsf{Q},\mathsf{R}), t \leq 1 \\ \phi(\mathsf{Q})|\phi(\mathsf{R}) & \mathsf{P} = \mathsf{Q}|\mathsf{R} \\ (\nu x)\phi(\mathsf{Q}) & \mathsf{P} = (\nu x)\mathsf{Q} \\ \mathsf{P} & \text{else} \end{cases}$$

Thus $\phi(\mathsf{P})$ ticks each timer in P by one discrete degree: this can be thought of as the passing of, say, one second in a global clock. Note that this function only acts on non-guarded timers: it does not influence timers under prefixes. This indicates a timer starts only after the guarding prefix is taken off, i.e. after the process is launched into the environment. Timers guarded by prefixes are said to be *inactive*, otherwise they are *active*. The free and bound names of timers are obtained by set-union from those of the timer's continuations.

Using this extended set of processes, the set of networks is defined just as in Section 3.3. This means that timers might be distributed among different sites. There can be two assumptions about how these timers would relate to each other: there would be a global clock, or time would be local to each site. Here, following Lamport's principle of local clocks [18], we take the second option. Synchronisation among local clocks in different sites can easily be incorporated, but this is not our concern in the present work. With this in mind, the dynamics of timers is given as follows.

$$
\begin{array}{rl}
(\textsc{TimeIn}) & \mathsf{timer}^{t+1}(x(\vec{v}).\mathsf{P},\mathsf{Q}) \mid \overline{x}\langle\vec{y}\rangle \;\longrightarrow\; \mathsf{P}\{\vec{y}/\vec{v}\} \\[1ex]
(\textsc{Par}) & \dfrac{\mathsf{P} \longrightarrow \mathsf{P}'}{\mathsf{P}|\mathsf{Q} \longrightarrow \mathsf{P}'|\phi(\mathsf{Q})} \\[2ex]
(\textsc{Idle}) & \mathsf{P} \;\longrightarrow\; \phi(\mathsf{P})
\end{array}
$$

The communication rule for timers with branching input is defined accordingly. The rules (TimeIn) and (Idle) are naturally extended to reduction in networks, though (Par) is extended to networks only in the sense that $\mathsf{M} \longrightarrow \mathsf{M}'$ implies $\mathsf{M}|\mathsf{N} \longrightarrow \mathsf{M}'|\mathsf{N}$, because we assume local clocks. The idea underlying the above reduction rules is that a timer which is not under some prefix *necessarily* advances whenever there is some reduction within the same site. The underlying intuition is that a reduction (computation) always takes some time: we represent it here as one discrete unit. But time can advance without computation happening, which is not only natural but also is essential if we wish to model timeouts when a process waits for an expected message but which may not come due to, say, message loss [4]. This is the purpose of the idle rule. The corresponding additions to the transition relation are as follows.

$$
\begin{array}{rl}
(\textsc{TimeIn}) & \mathsf{timer}^{t}(x(\vec{v}).\mathsf{P},\mathsf{Q}) \;\xrightarrow{x((\nu\vec{y})\vec{z})}\; \mathsf{P}\{\vec{z}/\vec{v}\} \\[1ex]
(\textsc{Par}) & \dfrac{\mathsf{P} \xrightarrow{\pi} \mathsf{P}' \qquad \mathsf{bn}(\pi)\cap\mathsf{fn}(\mathsf{Q})=\emptyset}{\mathsf{P}|\mathsf{Q} \;\xrightarrow{\pi}\; \mathsf{P}'|\phi(\mathsf{Q})} \\[2ex]
(\textsc{Idle}) & \mathsf{P} \xrightarrow{\tau} \phi(\mathsf{P})
\end{array}
$$

Using the extended set of processes, we incorporate sites and message loss just as before.

Immediately we obtain the weak bisimilarity $\approx$ over processes and over networks, for which we have the following result.

Proposition 4.1 *(i) $\approx$ and $\sim$ on processes are preserved by prefix and restriction but not by parallel composition. (ii) $\approx$ and $\sim$ are congruences over networks.*

As a simple example of the failure of congruence consider $\mathsf{P}^t = \mathsf{timer}^t(y.\overline{z}, 0)$. Then $\mathsf{P}^1 \approx \mathsf{P}^2$. But $x|\mathsf{P}^1$ is not bisimilar to $x|\mathsf{P}^2$ (since, while the input action by the former inevitably makes it 0, that by the latter can still retain P^1). See [3] for further discussions.

4.3 Extending the π-Calculus (3): Process Failure and Persistence

Message loss is not the only problem for distributed systems. Machines and processes in distributed systems can fail or crash. This is not specific to distributed systems: however, when a centralized system crashes, the whole computational process comes to an end. There is no notion of *partial failure* in centralized systems. On the other hand, one of the key characteristics of distributed systems is that they have a natural notion of partial failures and are supposed to tolerate this type of failure.

This section introduces partial failure at the level of sites. For a process to *fail* or *crash* (henceforth we shall use these two terms interchangeably) means that it cannot participate in interactions, and that it cannot itself reduce, until it restarts (which it might or might not do). We may conceive of crashed but not restarted processes as processes that, before restarting, act like the $\mathsf{0}$ process. We allow processes to fail and restart at any point in time. To be precise, we are assuming that failure cannot occur *during* an interaction, that is we assume the action of a process receiving a message to be atomic. This gives a fairly accurate abstraction of real distributed systems under the assumption that no Byzantine failure is possible. Failures and restarts of sites are completely non-deterministic events. There is nothing a process can do to influence failure or restart.

In order to cope with the possibility of site failures and restarts, distributed systems allow processes to specify how to restart, but not if or when. This can be achieved in many ways that often boil down to the existence of *persistent memory* that is unaffected by failures, together with mechanism that allows restarting processes to read data off that persistent store to find out as what kind of process it should reemerge. There are diverse ways to save state: operating systems often save entire processes for scheduling purposes at run time. A weaker mechanism would allow 'passive data' to be made persistent, for example indications that a process has passed a certain program point. Data that is made persistent to aid later restart is called a *savepoint*. We note that persistence and savepointing have received much attention from the distributed systems community [7], but to the best of our knowledge process theoretic accounts are lacking. While the mechanisms we present only skim the surface of this complex topic, we hope to offer a valuable starting point for further study.

The extensions with failure and restart mechanisms we use in the 2PCP are straightforward and can be built on top of the basic π-calculus or the calculus extended with message loss and timers. For convenience we opt for the latter. At the level of processes, we introduce a new prefix for making savepoints. As the π-calculus does not distinguish state, data and processes, we allow processes to be savepoints.

$$\mathsf{P} \quad ::= \quad \ldots \mid \mathsf{save}\langle \mathsf{P} \rangle.\mathsf{Q}.$$

The notion of process crash is incorporated at the level of sites. Each site records the latest savepoint in a superscript, with new savepoints overwriting previous ones. Additionally, we need to represent a crashed process that has not yet restarted. We denote such a process by $[\star]^{\mathsf{P}}_A$. Networks are now given by the following grammar.

$$\mathsf{N} \quad ::= \quad \ldots \mid [\mathsf{P}]^{\mathsf{Q}}_A \mid [\star]^{\mathsf{P}}_A$$

A process $[\mathsf{P}]^{\mathsf{Q}}_A$ should be understood as a a site containing a process P with the latest savepoint being Q. Should it crash and later restart, it will restart as Q (to be precise, it will reemerge as $[\mathsf{Q}]^{\mathsf{Q}}_A$). We do not require Q to have any resemblance to P. The well-formedness conditions are extended as follows. $\vdash [\mathsf{P}]^{\mathsf{Q}}_A$ if P as well as Q are local and have all their free input subjects in A. $\vdash [\star]^{\mathsf{P}}_A$ if P is local and all its free input subjects are in A. Finally, $\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{Q}$ is local if P and Q are and the free input subjects of $\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{Q}$ are the union of the free input subjects of P and Q.

A site $[\star]^{\mathsf{P}}_A$ represents a crashed process that will become $[\mathsf{P}]^{\mathsf{P}}_A$ should it ever restart. We overload the operators $|$, (νx) and 0 as before, and keep their algebraic laws. Other definitions including the free and bound names are standard. We set $\mathrm{fn}([\mathsf{P}]^{\mathsf{Q}}) = \mathrm{fn}(\mathsf{P}) \cup \mathrm{fn}(\mathsf{Q})$ and $\mathrm{fn}([\star]^{\mathsf{P}}) = \mathrm{fn}(\mathsf{P})$, and similarly for bound names. We need one additional axiom to define the structural congruence.

$$[(\nu x)\mathsf{P}]^{\mathsf{Q}}_A \equiv (\nu x)[\mathsf{P}]^{\mathsf{Q}}_{A \cup \{x\}} \qquad \text{if } x \notin \mathrm{fn}(\mathsf{Q})$$

For reduction, we add:

$$(\textsc{Save}) \ [\mathsf{P}|\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{R}]^{\mathsf{S}}_A \to [\mathsf{P}|\mathsf{R}]^{\mathsf{Q}}_A$$

$$(\textsc{Stop}) \ [\mathsf{P}]^{\mathsf{Q}}_A \to [\star]^{\mathsf{Q}}_A$$

$$(\textsc{Restart}) \ [\star]^{\mathsf{P}}_A \to [\mathsf{P}]^{\mathsf{P}}_A$$

$$(\textsc{Intra}) \ \frac{\mathsf{P} \to \mathsf{Q}}{[\mathsf{P}]^{\mathsf{R}}_A \to [\mathsf{Q}]^{\mathsf{R}}_A}$$

The (Stop) rule turns a process into a crash state while leaving its persistent storage. The saved state is used when restarting a process by Restart rule. The (Stop) and (Restart) rules allow network failure and recovery to happen asynchronously. We also note that the above dynamics assume that crashed processes can always restart: we consider only this case since standard treatment of the 2PCP is based on this idea. It is possible to have a variant in which some of crashed processes cannot restart

For the labelled semantics of the extended calculus we add labels of the from $\mathsf{save}\langle\mathsf{P}\rangle$ (where P is any process) to the core calculus. The free and bound names of the action $\mathsf{save}\langle\mathsf{P}\rangle$ are defined to be the free and bound names of

P. The transition system is then defined inductively by the following rules, in addition to those of the previous calculus.

$$(\text{S-Out})\ \mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{P} \xrightarrow{\mathsf{save}\langle\mathsf{Q}\rangle} \mathsf{P}$$

$$(\text{Restart})\ [\star]^{\mathsf{Q}}_{A} \xrightarrow{\tau} [\mathsf{Q}]^{\mathsf{Q}}_{A}$$

$$(\text{Save})\ \frac{\mathsf{P} \xrightarrow{\mathsf{save}\langle\mathsf{R}\rangle} \mathsf{Q}}{[\mathsf{P}]^{\mathsf{S}}_{A} \xrightarrow{\tau} [\mathsf{Q}]^{\mathsf{R}}_{A}}$$

$$(\text{Stop})\ [\mathsf{P}]^{\mathsf{Q}}_{A} \xrightarrow{\tau} [\star]^{\mathsf{Q}}_{A}$$

$$(\text{Proc})\ \frac{\mathsf{P} \xrightarrow{\pi} \mathsf{Q} \quad \pi \neq \mathsf{save}\langle\mathsf{S}\rangle}{[\mathsf{P}]^{\mathsf{R}}_{A} \xrightarrow{\pi} [\mathsf{Q}]^{\mathsf{R}}_{A}}$$

The (S-Out) rule introduces a form of process passing. It is weaker than process passing in higher order π-calculi [26] because no process can interact with an emission of $\mathsf{save}\langle\mathsf{P}\rangle$. Interaction with such an emission is exclusive to the persistence mechanism integrated into sites.

Definition 4.2 A symmetric binary relation of processes is a *bisimulation* if $\mathsf{P}\mathcal{R}\mathsf{Q}$ implies:

- whenever $\mathsf{P} \xrightarrow{\pi} \mathsf{P}'$, $\pi \neq \mathsf{save}\langle\mathsf{R}\rangle$, $\mathrm{bn}(\pi) \cap \mathrm{fn}(\mathsf{Q}) = \emptyset$ then we can find a transition $\mathsf{Q} \xrightarrow{\pi} \mathsf{Q}'$ such that $\mathsf{P}'\mathcal{R}\mathsf{Q}'$.
- whenever $\mathsf{P} \xrightarrow{\mathsf{save}\langle\mathsf{R}\rangle} \mathsf{P}'$, $\mathrm{bn}(\mathsf{save}\langle\mathsf{R}\rangle) \cap \mathrm{fn}(\mathsf{Q}) = \emptyset$ then we can find a transition $\mathsf{Q} \xrightarrow{\mathsf{save}\langle\mathsf{R}'\rangle} \mathsf{Q}'$ such that $\mathsf{P}'\mathcal{R}\mathsf{Q}'$ and $(\mathsf{R}, \mathsf{R}') \in \mathcal{R}$.

Bisimilarity on processes, denoted $\approx$ as usual, is the greatest bisimulation. Similarly, one defines *strong bisimilarity* on processes. Extensions of notions bisimulation and strong bisimulation to networks are straightforward, as saving induced process passing is not observable in networks.

Proposition 4.3 *Bisimilarity $\approx$ and strong bisimilarity $\sim$ are congruences on networks but not on processes.*

4.4 Description of 2PCP (2): the Full Protocol

Using the extended π-calculus, we can now rigorously describe the behaviour of the 2PCP incorporating all failure assumptions. As discussed, there are two stages where messages can be lost in the protocol: one is when votes are cast by sending them to from participants to the coordinator, the other is when the decision is communicated from the coordinator to the participants. Thus timers need be incorporated on both of these occasions to deal with message loss. Processes can fail at any point during the computation. This possibility is dealt with by adding savepoints before state changing decisions are externalized.

The whole configuration is given as follows. Each of the coordinator and participants are located in a separate site.

$$\mathsf{2PCP} = (\nu\vec{e})(\nu\vec{\mathsf{vote}})(\nu\vec{\mathsf{dec}})([\mathsf{C}(t_0)]_A^{\mathsf{S}_{false}} | [\mathsf{P}_1(t'_0)]_{A_1}^{\mathsf{P}_1^{false}} | \ldots | [\mathsf{P}_n(t'_0)]_{A_n}^{\mathsf{P}_n^{false}}).$$

where $C(t)$ and $P_i(t)$ are as given below. The variable t denotes the timeout periods of the timers in the respective processes. A suitable choice for t'_0 would be any number exceeding 1 while for t_0 one can choose any number greater than $n+1$. The access points are defined as in the previous section: $A = \{\vec{\mathsf{vote}}, \vec{e}\}$ and $A_i = \{\mathsf{dec}_i\}$. First we present the coordinator. Below and henceforth we use recursive equations of form $P = C[P]$ where P should always be under prefix, assuming the standard encoding [22] using replication. Now we consider the coordinator.

$$\begin{aligned}
\mathsf{C}(t) &= (\nu\vec{c})(\nu c_{self})(\nu a)(\nu \mathsf{vote}_{self})(\mathsf{C}_{wait}(t)|\mathsf{C}_{end}^{true}|\mathsf{C}_{end}^{false}|\mathsf{C}_{self}) \\
\mathsf{C}_{wait}(t) &= \mathsf{C}_{wait_1}(t)| \ldots |\mathsf{C}_{wait_n}(t)|\mathsf{C}_{wait_{self}} \\
\mathsf{C}_{wait_i}(t) &= \mathsf{timer}^t(\mathsf{vote}_i[\overline{c_i}, \overline{a}], \overline{a}) \\
\mathsf{C}_{self} &= \overline{\mathsf{vote}_{self}}\mathsf{left} \oplus \overline{\mathsf{vote}_{self}}\mathsf{right} \\
\mathsf{C}_{wait_{self}} &= \mathsf{vote}_{self}[\overline{c_{self}}, \overline{a}] \\
\mathsf{C}_i^{timein} &= \mathsf{vote}_i[\overline{c_i}, \overline{a}] \\
\mathsf{C}_{end}^{true} &= c_1. \ldots .c_n.c_{self}.\mathsf{save}\langle \mathsf{S}_{true}\rangle.\mathsf{S}_{true} \\
\mathsf{C}_{end}^{false} &= a.\mathsf{S}_{false} \\
\mathsf{S}_{true} &= \mathsf{S}_{true,1}| \ldots |\mathsf{S}_{true,n} \\
\mathsf{S}_{false} &= \mathsf{S}_{false,1}| \ldots |\mathsf{S}_{false,n} \\
\mathsf{S}_{true,i} &= \overline{\mathsf{dec}_i}\mathsf{left}|e_i.\mathsf{S}_{true,i} \\
\mathsf{S}_{false,i} &= \overline{\mathsf{dec}_i}\mathsf{right}|e_i.\mathsf{S}_{false,i}
\end{aligned}$$

We focus on four sub-behaviours, C_{wait_i}, $\mathsf{S}_{true,i}$, C_{end}^{true} and $\mathsf{S}_{false,i}$. C_{wait_i} waits for a vote from the i-th participant using a timeout, because the vote can be lost during transmission. This is crucial for the whole protocol not to deadlock. $\mathsf{S}_{true,i}$ is the subbehaviour in charge of sending the commit decision to the i-th participant. Since this decision message can also be lost, we let a participant request for a decision using timeout. Thus, for example, $\mathsf{S}_{true,i}$ first sends a commit decision then waits at e_i for the request to arrive from the i-th participant. If the request comes, it resends the same decision, and again waits at e_i. C_{end}^{true} checks if all cast votes are for committing. If they are, before externalising the command to participants to commit (S_{true}), it savepoints the result of the vote. Once this has been done, no further message loss or process failure can prevent the overall outcome of the protocol to be that all participants eventually commit. If the coordinator crashes before this savepoint has been taken, it can only restart as the aborting coordinator. This behaviour is achieved by the initial savepoint being S_{false}.

Now the participants become:

$$\begin{aligned}
\mathsf{P}_i(t) &= \mathsf{P}_i^{true}(t) \oplus \mathsf{P}_i^{false}\\
\mathsf{P}_i^{true}(t) &= \mathsf{save}\langle \mathsf{P}_i^{c}(t)\rangle.\mathsf{P}_i^{c}(t)\\
\mathsf{P}_i^{false} &= \overline{\mathsf{vote}_i}\mathsf{right}|!\overline{\mathsf{abort}_i}\\
\mathsf{P}_i^{c}(t) &= \overline{\mathsf{vote}_i}\mathsf{left}|\mathsf{P}'_i(t)\\
\mathsf{P}'_i(t) &= \mathsf{timer}^t(\mathsf{dec}_i[\mathsf{save}\langle !\overline{\mathsf{commit}_i}\rangle.!\overline{\mathsf{commit}_i}, \mathsf{save}\langle !\overline{\mathsf{abort}_i}\rangle.!\overline{\mathsf{abort}_i}],\ \overline{e_i}|\mathsf{P}'_i(t_0))
\end{aligned}$$

Each participant starts by non-deterministically deciding how to vote. With the initial savepoint being the aborting i^{th}-participant, only if the decision is to commit, this decision needs to be made persistent before it is communicated to the coordinator, otherwise, when waiting for the decision, it uses a timeout to cope with message loss: if a timeout takes place, it requests the decision again. Since a message loss (of either this request itself or the repeated vote) can take place again, the behaviour is given recursively, possibly asking for the decision unboundedly many times. If the message from the coordinator has successfully been communicated, an appropriate savepoint is taken, ensuring that subsequent crashed do not lead to additional interactions with the coordinator. This ensures the atomicity in the face of message loss and process failure because, essentially speaking, the decision of the coordinator is constant once it is determined: thus we may expect the same decision to be eventually communicated to each participant. The framework of reasoning about the resulting behaviour is presented in the next section, where we show how the central property of the 2PCP, atomicity, can be cleanly formulated and established for the above configuration.

5 Proving Atomicity

5.1 Atomicity in Processes

The process representation of protocols in particular and of computational structures in general would have two purposes: to precisely describe their operational behaviour and to analyse their behavioural properties based on the description. This section discusses how the process description of the 2PCP in the preceding section can be used for reasoning about *atomicity*, a key property of the 2PCP. As noted in Section 3, the atomicity property can be cleanly represented using bisimilarity, although the formulation of the property for the full protocol needs care due to the existence of sites and the possibility of site failures. The proof is done based on algebraic laws associated with the calculus extension. Many of these equations capture the significant interplay among timers, persistence and message loss in general forms.

Theorem 5.1 *Let* $\mathsf{Commit} = \Pi_{i=1}^{n}!\overline{\mathsf{commit}_i}$, *and* $\mathsf{Abort} = \Pi_{i=1}^{n}!\overline{\mathsf{abort}_i}$, $\mathsf{P}_c = \mathsf{save}\langle\mathsf{Commit}\rangle.\mathsf{Commit}$ *and* $\mathsf{P}_a = \mathsf{save}\langle\mathsf{Abort}\rangle.\mathsf{Abort}$.

(i) Assume that $\mathsf{P}_i \in \{\mathsf{P}_i^{\mathrm{true}}, \mathsf{P}_i^{\mathrm{false}}\}$ *for all* $i \in \{1,\ldots,n,\mathrm{self}\}$ *and* $\mathsf{P}_{i_0} = \mathsf{P}_{i_0}^{\mathrm{false}}$ *for some* $i_0 \in I$. *Then* $C_0[[\mathsf{P}_1]_{A_1}^{\mathsf{P}_1}]\ldots[[\mathsf{P}_n]_{A_n}^{\mathsf{P}_n}][\mathsf{P}_{\mathrm{self}}] \approx [\mathsf{Abort}]_{\emptyset}^{\mathsf{Abort}}$.

(ii) $C_0[[\mathsf{P}_1^{\text{true}}]_{A_i}^{\mathsf{P}_1^{\text{true}}}]\ldots[[\mathsf{P}_n^{\text{true}}]_{A_n}^{\mathsf{P}_n^{\text{true}}}][\mathsf{P}_{\text{self}}^{\text{true}}] \approx [\mathsf{P}_c \oplus \mathsf{P}_a]_{\emptyset}^{\mathsf{P}_c \oplus \mathsf{P}_a}$.

(iii) $\mathsf{2PCP} \approx [\mathsf{P}_c \oplus \mathsf{P}_a]_{\emptyset}^{\mathsf{P}_c \oplus \mathsf{P}_a}$.

The theorem asserts that the full 2PCP behaves, as far as external observers can tell, in one of the two ways: as the committing process or as the aborting process. Furthermore, if one or more of the participants or the coordinator decide to abort, then all participants will abort.

5.2 *Correctness Proof (1): Two Basic Equations*

The essential feature of our proof of Theorem 5.1 is that it is equational: a succession of smaller equations leads to the desired bisimilarity. Among them, we single out two because in addition to being basic for our present proof, they capture a fundamental relationship between message loss, timers and persistence. Their proofs, via appropriate closures, can be found in [4].

Message loss induces a certain type of non-determinism: a message may arrive or may be lost during transmission. Thus a natural idea is to simplify the equation by removing lossy messages and reduce them to non-deterministic choice at the receiver's side. This is indeed possible when combined with a suitable form of timers, as the following lemma demonstrates. The formulation needs care due to the interaction between timers and persistence. Below by *process reduction context* we mean a context whose hole is not under prefix and which does not contain networks. A process or a context is *timer-free* (*save-free*) if it does not contain timers (subexpressions of the form $\mathsf{save}\langle \mathsf{P}\rangle.\mathsf{Q}$).

Lemma 5.2 *Let C be a reduction context, C' be a timer-free process reduction context, the set I be partitioned by I_l, I_r, and $x_i \notin fn(\mathsf{P}_j, \mathsf{R}, C, C') \cup A_j \cup B \cup \{y_j, z\}$ for all $i, j \in I$. Then, with $\mathsf{S}_i = \Pi_{i\in I}\mathsf{timer}^{t_0}(x_i[\overline{y_i}, \overline{z}], \overline{z})$, we have:*

$$(\nu\vec{x})C[\ \Pi_{i\in I_l}[\overline{x_i}\mathsf{left} \mid \mathsf{P}_i]_{A_i}^{\overline{x_i}\mathsf{left} \mid \mathsf{P}_i} \mid \Pi_{i\in I_r}[\overline{x_i}\mathsf{right} \mid \mathsf{P}_i]_{A_i}^{\overline{x_i}\mathsf{right} \mid \mathsf{P}_i} \mid [C'[\ \mathsf{S}_i\ ]]_A^{\mathsf{R}}\ ]$$
$$\approx C[\ \Pi_{i\in I_l}[\mathsf{P}_i]_{A_i}^{\mathsf{P}_i} \mid \Pi_{i\in I_r}[\mathsf{P}_i]_{A_i}^{\mathsf{P}_i} \mid [C'[\ \Pi_{i\in I_l}\overline{y_i} \oplus \overline{z} \mid \Pi_{i\in I_r}\overline{z}\ ]]_A^{\mathsf{R}}\ ].$$

Note how branching works effectively. Below, in Lemmas 5.4 and 5.5, this equation is used to reduce the possible loss of message of votes to nondeterminism on the part of the coordinator.

The next equation concerns the use of *recursive timers*. A recursive timer is another form of the use of timers in distributed algorithms. The objective is to cancel the effect of partial failures such as message loss and site crash by repeated actions based on timeouts. As such, there should be a general equation which precisely captures this effect. The following lemma gives one, indicating how the effect of message loss can be ignored by a certain forms of use of recursive timers.

Lemma 5.3 *Assume C is a reduction context. Assume that $I \subseteq J$. Also let $\mathsf{S}_i(t)$ be such that $\mathsf{S}_i(t) = \mathsf{timer}^t(x_i[\mathsf{P}_i, \mathsf{Q}_i], \overline{e_i}|\mathsf{S}_i(t_0))$, $\mathsf{Q}_i = \mathsf{save}\langle \mathsf{R}_i\rangle.\mathsf{R}_i$,*

$\mathsf{T}_i = \overline{x_i}\mathsf{right}|e_i.\mathsf{T}_i$ *with, in each case,* t *is such that* $0 < t \leq t_0$ *and* $t_0 > 1$. *Assume further* $\{x_i, e_i\}_{i\in I} \cap fn(C, \mathsf{R}_k, \mathsf{T}_l) = \emptyset$ *for all* $k \in I$ *and all* $l \in J \backslash I$. *Then* $(\nu\{x_i, e_i\}_{i\in I})C[\ \Pi_{i\in I}[\mathsf{S}_i(t)]_{A_i}^{\mathsf{S}_i(t_0)} \mid [\Pi_{i\in J}\mathsf{T}_i]_B^{\Pi_{i\in J}\mathsf{T}_i}\ ] \approx C[\ \Pi_{i\in I}[\mathsf{R}_i]_{A_i}^{\mathsf{R}_i} \mid [\Pi_{i\in J\backslash I}\mathsf{T}_i]_B^{\Pi_{i\in J\backslash I}\mathsf{T}_i}\ ]$. *There is a symmetric version with* $\overline{x_i}\mathsf{left}$ *instead of* $\overline{x_i}\mathsf{right}$.

The equation has a natural counterpart for non-branching prefixes and is used below for eliminating recursive timers in participant waiting for the coordinator's decision.

5.3 Correctness Proof (2): Main Part

We are now ready to embark on the outline of the proof of Theorem 5.1. Its proof is split in three parts: Lemmas 5.4 and 5.5 establish the content of Theorem 5.1 assuming that all decisions have already been made. These results are then combined using Lemma 5.6 (iv). The proofs of Lemmas 5.4 and 5.5 proceed by algebraic reasoning using equations established in Lemmas 5.2, 5.3 and 5.6. To aid legibility we highlight those parts of a given network that are changed in each step. We offer the algebraic reasoning to establish these results in some detail. The equations in Lemma 5.6 are mostly tailor made to meet the needs of the proofs of Lemmas 5.4 and 5.5 and are placed at the end.

Lemma 5.4 *Assume that* $n > 0$ *and for all* $i \in \{1, \ldots, n\}$ $\mathsf{N}_i = [\mathsf{P}_i]_{A_i}^{\mathsf{P}_i}$ *and for all* $i \in \{1, \ldots, n, \mathrm{self}\}$ *it is the case that* $\mathsf{P}_i \in \{\mathsf{P}_i^{\mathrm{true}}, \mathsf{P}_i^{\mathrm{false}}\}$, *but for at least one appropriate* $i_0, \mathsf{P}_{i_0} = \mathsf{P}_{i_0}^{\mathrm{false}}$. *Then* $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{\mathrm{self}}] \approx [\mathsf{Abort}]_{\emptyset}^{\mathsf{Abort}}$.

Proof. We shall assume that $\mathsf{P}_{self} = \overline{\mathsf{vote}_{self}}\mathsf{left}$. The case that $\mathsf{P}_{self} = \overline{\mathsf{vote}_{self}}\mathsf{right}$ is dealt with similarly. Now, $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{self}]$ is structurally equivalent to

$$
\begin{aligned}
&C[\ \Pi_{i\in I_c}[\ \overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{P}_i(t'_0)\ ]_{A_i}^{\overline{\mathsf{vote}_i}\mathsf{left}\ \mid \mathsf{P}_i(t'_0)}\\
&\mid \Pi_{i\in I_a}[\ \overline{\mathsf{vote}_i}\mathsf{right} \mid !\overline{\mathsf{abort}_i}\ ]_{A_i}^{\overline{\mathsf{vote}_i}\mathsf{right}\ \mid !\overline{\mathsf{abort}_i}}\\
&\mid [C'[\ \overline{\mathsf{vote}_{self}}\mathsf{left} \mid \Pi_{i=1}^n \mathsf{Q}_i \mid \mathsf{vote}_{self}[\overline{c_{self}}, \overline{a}]\ \mid c_1 \ldots c_n.c_{self}.\mathsf{S}_{true} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]
\end{aligned}
$$

where $\mathsf{Q}_i = \mathsf{timer}^{t'}(\mathsf{vote}_i[\overline{c_i}, \overline{a}], \overline{a})$. Applying Lemma 5.2 and Lemma 5.6 (viii) and (v), gives

$$
\begin{aligned}
&\approx C[\ \Pi_{i\in I_c}[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}\\
&\mid [C'[\ \Pi_{i\in I_c}\overline{c_i} \oplus \overline{a} \mid \overline{c_{self}} \mid \Pi_{i\in I_a}\overline{a} \mid c_1 \ldots c_n.c_{self}.\mathsf{S}_{true} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]
\end{aligned}
$$

Next, we use Lemma 5.6 (vii) and (v).

$$
\begin{aligned}
&\approx C[\ \Pi_{i\in I_c}[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}\\
&\mid [C'[\ \Pi_{i\in I_c}\overline{c_i} \oplus \overline{a} \mid c_1 \ldots c_n.c_{self}.\mathsf{S}_{true} \mid \mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]
\end{aligned}
$$

As $I_a \neq \emptyset$, we can find some $i_0 \in \{1, \ldots, n, \mathit{self}\}$: such that $i_0 \notin I_c$, so $c_{i_0} \notin \mathrm{fn}(C') \cup \mathrm{fn}(\mathsf{S}_{\mathit{false}}) \cup A$. Hence we can apply Lemma 5.6 (v) and (viii) to obtain

$$\begin{aligned}&\approx C[\ \Pi_{i\in I_c}[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \\ &\ \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid [\ C'\ [\ \Pi_{i\in I'_a}\overline{a} \mid a.\mathsf{S}_{\mathit{false}}\ ]]_A^{\mathsf{S}_{\mathit{false}}}\ ] \\ &\equiv C[\ \Pi_{i\in I_c}[\ \mathsf{P}_i(t'_0)\ ]_{A_i}^{\mathsf{P}_i(t'_0)} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid\ [\mathsf{S}_{\mathit{false}}]_A^{\mathsf{S}_{\mathit{false}}}\ ]\end{aligned}$$

Now we use Lemma 5.3 and Proposition 4.3 where $\mathsf{S}'_{\mathit{false}} = \Pi_{i\in I_a}\mathsf{S}_{\mathit{false},i}$.

$$\approx C[\ \Pi_{i\in I_c}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid\ [\mathsf{S}'_{\mathit{false}}]_{'}^{\mathsf{S}_{\mathit{false}}} A\ ]$$

Next apply Lemma 5.6 (iii)

$$\approx C[\ \ \Pi_{i\in I_c}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}\ \ ] \equiv C[\ \Pi_{i=1}^n[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}\ ]$$

We then use Lemma 5.6 (ii) to get

$$\approx\ C\ [\ \Pi_{i=1}^n[!\overline{\mathsf{abort}_i}]_\emptyset^{!\overline{\mathsf{abort}_i}}\ ] \equiv\ \Pi_{i=1}^n[!\overline{\mathsf{abort}_i}]_\emptyset^{!\overline{\mathsf{abort}_i}}$$

By Lemma 5.6 (i) this gives

$$\approx [\Pi_{i=1}^n !\overline{\mathsf{abort}_i}]_\emptyset^{\Pi_{i=1}^n !\overline{\mathsf{abort}_i}} = [\mathsf{Abort}]_\emptyset^{\mathsf{Abort}},$$

as required. □

Lemma 5.5 *Let $n > 0$, and $\mathsf{N}_i = [\mathsf{P}_i^{\mathrm{true}}]_{A_i}^{\mathsf{P}_1^{\mathrm{true}}}$ then $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{\mathrm{self}}^{\mathrm{true}}] \approx [\mathsf{P}_a \oplus \mathsf{P}_c]_\emptyset^{\mathsf{P}_a\oplus\mathsf{P}_c}$.*

Proof. $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{\mathit{self}}^{\mathit{true}}]$ is structurally equivalent to

$$\begin{aligned}&C[\ \Pi_{i=1}^n[\ \overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{P}_i(t'_0)\ ]_{A_i}^{\overline{\mathsf{vote}_i}\mathsf{left}\ \mid\ \mathsf{P}_i(t'_0)} \\ &\ \mid [C'[\ \overline{\mathsf{vote}_{\mathit{self}}}\mathsf{left} \mid \Pi_{i=1}^n\mathsf{Q}_i \mid \mathsf{vote}_{\mathit{self}}[\overline{c_{\mathit{self}}}, \overline{a}] \mid c_1\ldots c_n.c_{\mathit{self}}.\mathsf{S}_{\mathit{true}} \mid a.\mathsf{S}_{\mathit{false}}\ ]]_A^{\mathsf{S}_{\mathit{false}}}\ ]\end{aligned}$$

where $\mathsf{Q}_i = \mathsf{timer}^{t'}(\mathsf{vote}_i[\overline{c_i}, \overline{a}], \overline{a})$. We apply Lemma 5.2 and Lemma 5.6 (v) to obtain

$$\approx C[\ \Pi_{i=1}^n[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \mid [C'[\ \ \Pi_{i=1}^n\overline{c_i} \oplus \overline{a} \mid \overline{c_{\mathit{self}}} \mid c_1\ldots c_n.c_{\mathit{self}}.\mathsf{S}_{\mathit{true}} \mid a.\mathsf{S}_{\mathit{false}}\ \ ]]_A^{\mathsf{S}_{\mathit{false}}}\ ]$$

Next we apply Lemma 5.6 (ix) and (v).

$$\begin{aligned}&\approx C[\ \Pi_{i=1}^n[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \mid [\ C'\ [\ \mathsf{save}\langle\mathsf{S}_{\mathit{true}}\rangle.\mathsf{S}_{\mathit{true}} \oplus \mathsf{S}_{\mathit{false}}\ ]]_A^{\mathsf{S}_{\mathit{false}}}\ ] \\ &\equiv C[\ \Pi_{i=1}^n[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \mid [\mathsf{save}\langle\mathsf{S}_{\mathit{true}}\rangle.\mathsf{S}_{\mathit{true}} \oplus \mathsf{S}_{\mathit{false}}]_A^{\mathsf{S}_{\mathit{false}}}\ ]\end{aligned}$$

To finish the proof, we will show that

$$C[\ \Pi_{i=1}^{n}[\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{save}\langle \mathsf{S}_{true}\rangle.\mathsf{S}_{true}]_A^{\mathsf{S}_{false}}\ ] \approx [\mathsf{P}_c \oplus \mathsf{P}_a]_\emptyset^{\mathsf{P}_c\oplus\mathsf{P}_a} \tag{1}$$

$$C[\ \Pi_{i=1}^{n}[\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ] \approx [\Pi_{i=1}^{n}!\overline{\mathsf{abort}_i}]_\emptyset^{\Pi_{i=1}^{n}!\overline{\mathsf{abort}_i}} \tag{2}$$

and then apply Lemma 5.6 (x). To establish Equation (2) we proceed as in the latter parts of the proof of Lemma 5.4, taking $I_a = \emptyset$. For (1) we observe that Lemma 5.6 (xi) guarantees that

$$C[\ \Pi_{i=1}^{n}[\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{true}]_A^{\mathsf{S}_{true}}\ ] \approx [\Pi_{i=1}^{n}!\overline{\mathsf{commit}_i}]_\emptyset^{\Pi_{i=1}^{n}!\overline{\mathsf{commit}_i}} \tag{3}$$

$$C[\ \Pi_{i=1}^{n}[\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ] \approx [\Pi_{i=1}^{n}!\overline{\mathsf{abort}_i}]_\emptyset^{\Pi_{i=1}^{n}!\overline{\mathsf{abort}_i}} \tag{4}$$

together imply Equation (3). To establish (3) we proceed as follows. Consider

$$C[\ \Pi_{i=1}^{n}[\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{true}]_A^{\mathsf{S}_{true}}\ ].$$

This network is bisimilar to the following, according to Lemma 5.3 together with Proposition 4.3.

$$\approx C[\ \Pi_{i=1}^{n}[!\overline{\mathsf{commit}_i}]_{A_i}^{!\overline{\mathsf{commit}_i}} \mid [\mathsf{S}_{true}]_A^{\mathsf{S}_{true}}\ ]$$

And then we apply Lemma 5.6 (iii).

$$\approx C[\ \Pi_{i=1}^{n}[!\overline{\mathsf{commit}_i}]_{A_i}^{!\overline{\mathsf{commit}_i}}\ ]$$

We can now use Lemma 5.6 (ii) to obtain

$$\approx C[\ \Pi_{i=1}^{n}[!\overline{\mathsf{commit}_i}]_\emptyset^{!\overline{\mathsf{commit}_i}}\ ] \equiv \Pi_{i=1}^{n}[!\overline{\mathsf{commit}_i}]_\emptyset^{!\overline{\mathsf{commit}_i}}.$$

Finally, Lemma 5.6 (i) allows to conclude

$$\approx [\Pi_{i=1}^{n}!\overline{\mathsf{commit}_i}]_\emptyset^{\Pi_{i=1}^{n}!\overline{\mathsf{commit}_i}} = [\mathsf{Commit}]_\emptyset^{\mathsf{Commit}}.$$

The proof of (4) is similar. □

Finally we present the remaining algebraic laws used above.

Lemma 5.6 *(i) Let $I \neq \emptyset$. Then $\Pi_{i\in I}[!\overline{x_i}]_\emptyset^{!\overline{x_i}} \approx [\Pi_{i\in I}!\overline{x_i}]_\emptyset^{\Pi_{i\in I}!\overline{x_i}}$.*

(ii) If $x_i \notin A$ for all $i \in I$, then $C[\ [\Pi_{i\in I}!\overline{x_i}]_A^{\Pi_{i\in I}!\overline{x_i}}\ ] \approx C[\ [\Pi_{i\in I}!\overline{x_i}]_\emptyset^{\Pi_{i\in I}!\overline{x_i}}\ ]$.

(iii) If $\{\vec{x}\} = fn([\mathsf{P}]_A^{\mathsf{Q}})$ and $\{\vec{x}\} \cap fn(C, \mathsf{N}) = \emptyset$ then $(\nu\vec{x})C[\ \mathsf{N} \mid [\mathsf{P}]_A^{\mathsf{Q}}\ ] \approx C[\ \mathsf{N}\ ]$.

(iv) Let $C[\ ]\ldots[\ ]$ be an $n+1$-ary reduction context taking n networks and (as rightmost argument) a process. Let $\mathsf{P}_i^j, \mathsf{R}_i$ be processes for $i = 1, \ldots, n+1$ $(n > 0)$and $j = 1, 2$ such that

$C[\ [\mathsf{P}_1^{i_1}]_{A_1}^{\mathsf{P}_1^{i_1}}\ ]\ldots[\ [\mathsf{P}_n^{i_n}]_{A_n}^{\mathsf{P}_n^{i_n}}\ ][\ \mathsf{P}_{n+1}^{i_{n+1}}\ ] \approx [\mathsf{P}]_A^{\mathsf{P}}$ where at least for one $j \in \{1, \ldots, n+1\} : i_j = 2$.

$C[\ [\mathsf{P}^1_1]^{\mathsf{P}^1_1}_{A_1}\ ]\ldots[\ [\mathsf{P}^1_n]^{\mathsf{P}^1_n}_{A_n}\ ][\ \mathsf{P}^1_{n+1}\ ] \approx [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}]^{\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}}_A$.
Then $C[\ [\mathsf{P}^1_1\oplus\mathsf{P}^2_1]^{\mathsf{P}^2_1}_{A_1}\ ]\ldots[\ [\mathsf{P}^1_n\oplus\mathsf{P}^2_n]^{\mathsf{P}^2_n}_{A_n}\ ][\ \mathsf{P}^1_{n+1}\oplus\mathsf{P}^2_{n+1}\ ] \approx [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}]^{\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}}_A$

(v) If $\mathsf{P}\approx\mathsf{Q}$ *then* $[\mathsf{P}]^{\mathsf{R}}_A \approx [\mathsf{Q}]^{\mathsf{R}}_A$.

(vii) Let $I\neq\emptyset$, Q *be timer-free and* C *be a timer-free process reduction context such that* $x\notin fn(C,\mathsf{P},\mathsf{Q})$. *Then* $(\nu x)C[\ \Pi_{i\in I}\overline{x}\mid x.\mathsf{P}\mid\mathsf{Q}\ ]\approx C[\mathsf{P}|\mathsf{Q}]$.

(viii) Let C *be a timer-free process reduction context such that* $\{x_1,\ldots,x_n,y\}\cap fn(C,\mathsf{P},\mathsf{Q})=\emptyset$. *Furthermore, assume that* $I\subset\{1,\ldots,n\}$ *and* $n>0$. *Then*

$$(\nu y)(\nu x_1)\ldots(\nu x_n)C[\ \Pi_{i\in I}\overline{x_i}\oplus\overline{y}\mid x_1\ldots x_n.\mathsf{P}\mid\mathsf{Q}\ ]\approx C[\mathsf{Q}].$$

(ix) Assume that $\{\vec{x},y\}\cap fn(C,\mathsf{P},\mathsf{Q})=\emptyset$, C *is a timer-free process reduction context and* $n>0$, *then* $(\nu\vec{x})(\nu y)C[\ \Pi^n_{i=1}\overline{x_i}\oplus\overline{y}\mid x_1\ldots x_n.\mathsf{P}\mid y.\mathsf{Q}\ ]\approx C[\ \mathsf{P}\oplus\mathsf{Q}\ ]$

(x) Assume that C *is a reduction context such that*

$$C[\ [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}]^{\mathsf{Q}}_A\ ]\approx[\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}]^{\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}}_B$$
$$C[\ [\mathsf{Q}]^{\mathsf{Q}}_A\ ]\approx[\mathsf{R}]^{\mathsf{R}}_B.$$

Then $C[\ [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{Q}]^{\mathsf{Q}}_A\ ]\approx[\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}]^{\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}}_B$

(xi) Assume that C *is a reduction context such that* $C[\ [\mathsf{P}]^{\mathsf{P}}_A\ ]\approx[\mathsf{R}]^{\mathsf{R}}_B$, $C[\ [\mathsf{Q}]^{\mathsf{Q}}_A\ ]\approx[\mathsf{S}]^{\mathsf{S}}_B$ *Then* $C[\ [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}]^{\mathsf{Q}}_A\ ]\approx[\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}]^{\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}}_B$.

6 Discussion

The extensions to the π-calculus we have introduced in the present paper, message loss, timers and process failure/recovery mechanisms, enable us to concisely represent and reason about the 2PCP, a realistic distributed algorithm. The simplicity of the description and the equational reasoning using the extensions suggest their possible applicability for other distributed algorithms. Below we offer further remarks on these constructs, first regarding their translatability into name passing and, secondly, regarding related works and further issues.

6.1 Translation into the π-Calculus

Given the expressive power of name passing interaction, a natural question is whether added constructs can be represented in the original π-calculus. A few essential points are involved in this question. First, the representability of a certain construct does *not* mean it is representable when combined with other added constructs: the interplay among diverse syntactic constructs is crucial. As a simple example, we take the known encoding (cf. [15,23]) of branching into the π-calculus:

$$[\![\overline{x}\mathsf{left}\langle\vec{v}\rangle]\!] \stackrel{\text{def}}{=} (\nu c)(\overline{x}\langle c\rangle | c(z_1 z_2).\overline{z_1}\langle\vec{v}\rangle)$$

(symmetrically for the right selection, and dually for the branching input). This encoding is sound and compositional in the base calculus without branching: however its natural encoding into the full calculus (or the calculus with message loss and timers) is much more involved than the above, since we need to consider the case when this message is sent remotely: we want *either* this message to be lost completely, *or* to be sent safely, in other works: atomically. For this purpose we need to use recursive timers just as we did for the two-phase commitment protocol. Thus it *is* possible to encode this branching construct into the full calculus, but its representation becomes quite complex. This example suggests that individual representability of extensions may not lead to the representability of their combination so that the study of expressiveness for these constructs should be performed with care.

With this in mind, we are currently studying the relative expressiveness of the extended constructs. Message loss can be encoded by translating a site as a collection of forwarders of a certain kind [10,15]. Essentially a site is translated into a system which takes care of all messaging from and to a site. On the other hand, timers are provably [3] not encodable fully abstractly and compositionally. We doubt that there is even a non-trivial sound compositional encoding (there is compelling evidence that such an encoding is hard to obtain, even though there seems to be a non-compositional way to represent global timing using specific messages representing signals) and even if we have, we may not be able to obtain the equations on timers as used in Section 5. For site-failure, a crucial point is that processes in a site should crash together, rather than partially. For this purpose, again the implementation in the basic π-calculus should have an elaborate operational structure, which is polled by processes for the state of crash each time it takes any action. The resulting construction is already quite complex: it remains to be seen how the structure can be combined with that of timers and other constructs, and what equational properties this encoding owns. As far as the constructions we have studied go, the translations first of all often do not enjoy satisfactory equational properties (such as compositionality and fully abstractness) and, second, they hardly assist our reasoning on these constructs and the concerned phenomena, especially when timers are involved. We believe further study on (im)possibility of satisfactory encodings would help us understand the status and nature of added constructs. Some aspects of this topic will be further discussed in the forthcoming [3].

6.2 Related Work and Further Issues

Process algebras have been used as syntactic tools for representing various computational phenomena involving concurrency and communication, and, as such, there have been many studies on the incorporation of "non-standard" features into process calculi. For example, the addition of (real-)time to process algebra has been studied extensively (see [13] for a survey). There are also recent studies on extensions of the π-calculus to describe various aspects of distributed systems, cf. [2,9,25]. In comparison with these, the present work differs in that it demonstrates how these constructs can be coherently combined semantically to offer a unified reasoning framework. As we saw in Section 5, the clear articulation of related phenomena and their combination in the form of timers, their localisation via sites, process crash and persistence, are crucial for reasoning about distributed algorithms, which would be, to our knowledge, first demonstrated in the present work. We also note that the use of the π-calculus as our base calculus is not arbitrary: distributed software consists of not only protocols but also programming languages (indeed protocols *are* implemented by programming languages). By using the π-calculus as our base language, a resulting formalism would be able to capture not only the part of protocols but the whole of distributed systems on a uniform basis.

An interesting further topic is the deeper study of the semantic properties of the extended calculus. In particular, timers drastically change the semantic theory. Not only is strong and weak bisimilarity no longer preserved by parallel composition (as noted in Section 3), but also strong and weak reduction-based congruence [8,17] coincide [3]. This is a consequence of the implicit synchronisation that timing engenders (see also [13] for discussion in a different setting). We conjecture that all reasonable congruences for the π-calculus with timers will exhibit similar properties. This suggests that conventional definitions of process equivalences are inappropriate in a timed setting. In [3], we present techniques to finely control the sensitivity of equivalences to timing. Nevertheless, much work in this area needs to be done, especially with regards to the combination of timers, message loss and process failure.

Another important topic is how additional syntactic constructs, as introduced in the present paper, can be systematically explored and developed, both at the syntactic and semantic level. What general concepts underly various notions which arise in distributed systems and which defy straightforward representation in the bare π-calculus? Relatedly, ramification of the introduced primitives is an interesting subject of study. As an example, the proposed process failure and recovery may be given a more general formulation. The presented constructs are sufficiently powerful to describe 2PCP; however they would be too simple for modeling diverse realistic process recovery mechanisms, cf. [7]. We wish to address these and related problems in future studies.

Finally, important remaining work is to take the present study further in representing other distributed algorithms and models. In particular, we are currently working on the representation of the more sophisticated atomic commitment protocols, such as the Three Phase Commitment Protocols [6], using the present extended calculus. Such study may also contribute to the clarification of basic building blocks and structures of this and other classes of distributed algorithms.

Acknowledgements
We thank Chris Hankin and the referees for their helpful comments.

References

[1] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Journal of Information and Computation*, 127(2):91–101, 1996.

[2] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. of COORDINATION 97*, volume 1282 of *LNCS*. Springer Verlag, Berlin, 1997. Also Rapport Interne 216 LIM February 1997, and INRIA Research Report 3109.

[3] Martin Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, Department of Computing, 2000. To appear.

[4] Martin Berger and Kohei Honda. Atomic commitment protocols in extended π-calculi (1). Available upon request from the authors, May 2000.

[5] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes. *TCS*, 37:77–121, 1985.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Elmootazbellah N. Elnozahy, David B. Johnson, and Yi-Min Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[8] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of ICALP 1998*, 1998.

[9] Cédric Fournet, Georges Gonthier, Jean-Jaques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Principles of Programming Languages*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, January 1996. Springer-Verlag.

[10] Cedric Fournet and Cosimo Laneve. Bisimulations in the join-calculus. To appear in *Theoretical Computer Science*.

[11] Jim Gray. Notes on data base operating systems, 1979.

[12] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[13] Matthew Hennessy. Timed process algebras: a tutorial, 1992.

[14] Charles Anthony Richard Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[15] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.

[16] Kohei Honda and Mario Tokoro. A small calculus for concurrent objects. *OOPS Messenger*, 1991.

[17] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151:437–486, 1995.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[19] Leslie Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[20] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. In *Fifth ACM Symposium on Operating Systems Principles*, pages 19–21, November 1975.

[21] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[22] Robin Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[23] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[24] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS'98*, 1998.

[25] James Riely and Matthew Hennessy. Distributed processes and location failures. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *lncs*, pages 471–481. Springer Verlag, Berlin, 1997.

[26] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms.* PhD thesis, University of Edinburgh, 1992.

[27] Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Programming Languages and Systems*, 1(3):222–238, 1983.

[28] Dale Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 133–142, 1981.

[29] P. Spiro, A. Joshi, and T. Rengarajan. Designing an optimized transaction commit protocol. *Digital Technical Journal*, 3(1), 1991.

[30] Vasco Vasconcelos. Typed concurrent objects. In *Proceedings of ECOOP'94*, pages 100–117. Springer Verlag, 1994.