Abstractions for Fault Tolerant Global Computing (Extended Abstract)

Dominic Duggan [1]

Department of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07030.

Abstract

Global computing (WAN programming, Internet programming) distinguishes itself from local computing (LAN computing) by among other things the fact that it exposes the network to the application, rather than seeking to hide it with network transparency as in LAN programming. Global computing languages seek to provide useful abstractions for building applications in such environments. This paper introduces the pik-calculus, a calculus for asynchronous distributed programming that incorporates abstractions for building fault-tolerant global applications. The calculus incorporates notions of atomic failure and failure dependencies, from which various forms of distributed transactions and optimistic computation may be built. The pik-calculus extends the asynchronous pi-calculus with a notion of logs and "safe" operations for modifying those logs.

1 Introduction

Global computing, sometimes referred to as wide-area computation, wide-area network programming or Internet programming [10], poses interesting challenges for application developers. This is because the traditional programming environments for distributed application development are based on applications spanning local-area networks and enterprise intra-networks. The characteristics of local computing environments are different from those of global computing, and suggest a need for different approaches.

Local computing languages are organized around the principle of location transparency, hiding the network behind the abstraction of RPC or RMI [6]. In contrast global computing languages expose the network to the application, recognizing that network management and navigation are important aspects of global applications,

[1] Email: dduggan@cs.stevens-tech.edu

and seeking to provide useful high-level abstractions for application developers. A commercial example of this is given by the Java Jini system [4], while a formal approach is given by the ambient calculus [11].

Work on the semantics of global computing languages has focused on mobility of various kinds [21,11,12,42,29]. There has been little attention paid to providing support for fault tolerance, aside from work based on fail-stop failure models, that may not always be appropriate in global computing [10]). An example of a local computing language that provided support for fault tolerance is the Argus language [34]. Fault tolerance was based on guardians and nested transactions [37,35]. Similar support for transactions was provided by languages such as Avalan/C++ and Venari/ML [17,28], and is an integral part of various well-known distributed computing platforms, including CORBA OTS, COM MTS, and Java Jini and JavaBeans [47,13,4].

There are two aspects of transactions, as a tool for building fault-tolerant global applications, that we wish to address:

(i) The first aspect is the somewhat monolithic concept of transactions themselves. Transactions include notions of failure atomicity, concurrency control, persistence, and undoing of effects. This particular combination is useful for the kinds of database applications for which transactions were originally designed. It is not clear that this particular combination, or any particular combination, is appropriate for all global applications. For example there are many variants of transactions that have been proposed for various other classes of applications, particularly for long-lived applications [19].

In this paper we propose a set of largely orthogonal abstractions, that can be combined to build different classes of fault-tolerant applications. Transactions are one of the mechanisms that can be built through such a combination. Fig. 1 provides our set of abstractions.

(ii) At the heart of mechanisms for building fault-tolerant applications are some collections of protocols, that in turn rely on a communication system for delivering protocol messages. In global computing, establishing communication channels may itself be an important part of a global application. For example Internet communication must nowadays navigate through firewalls, proxies, network address translators and load balancers. Currently this is done in an *ad hoc* fashion, in a way that violates data abstraction and the supposed layering of protocols. The ambient calculus is based on the notion of applications explicitly navigating administrative domains delimited by firewalls. If fault-tolerance protocols are part of the underlying support for building global applications, how are the protocol messages to be delivered in a semantically correct (and secure) manner?

In this paper we make some progress towards providing an answer to this issue, by isolating the communication requirements of protocols in the semantics of the underlying abstractions for building fault-tolerant global applications. There is a notion of stable storage and logs; processes may query

and extend their own local logs, but may only query the logs of remote sites. The issue of how to query the logs of a remote site, without relying on an underlying communication infrastructure, is left to a sequel paper.

The Jini system's [4] support for fault tolerance provides interfaces for the two-phase commit protocol [5], and a default implementation of the protocol. We take the position that this approach is too high level and too protocol-specific. It is too high level in the sense that atomic commitment is in general impossible to achieve in asynchronous distributed systems [20,27]. On the other hand, building a particular (potentially blocking) protocol for atomic commitment into the language semantics overcommits the language design to particular implementations. Our emphasis is instead on isolating the invariants that should be maintained by stable storage, and providing mechanisms that applications can use to change stable storage in such a way that these invariants are preserved. Protocols such as two-phase commit can then be provided as libraries on top of these primitives. As such, our approach is akin to a type system for a global programming environment, albeit one where the invariant-preserving operations on stable storage are checked at runtime. This is represented in our calculus by operations for appending to logs, that require preconditions to be satisfied before such an appending is allowed.

At the core of the abstractions in Fig. 1 is the concept of *atomic failures*. Atomic failures are achieved by grouping processes into process groups; we refer to such a process group as a *conclave*. A conclave is at its simplest level a group of processes

$$c\{P_1 \mid \ldots \mid P_k\}$$

where the conclave identifier c serves as a name for the group. The intention is that if any process in a conclave "fails," then all of the processes in that conclave "fail." We identify *causality* as the fundamental yardstick for measuring dependencies between the failures of different conclaves, and we propose *causal consistency* as a correctness criteron for proper executions involving atomic failures. If a conclave c_1 consumes some of the output of another conclave c_2, this establishes a causal dependency from c_2 to c_1. If c_2 subsequently aborts, then c_1 must also abort. The intuition of causal consistency is that, as with traditional transactions, a run of conclaves with failures should be in some sense equivalent to one in which no failures occurred, or in which at least no effects besides failure are visible for the failed conclaves (the latter alternative is possible with nested transactions). By "visible" we mean informally that no database updates or messages issued by failed processes are observed by processes outside the corresponding conclaves. Rather than restricting causal relationships to tree structures, we allow arbitrary directed graph structures, including cyclic graphs.

We refer to the calculus introduced here as the pik-calculus. In Sect. 2 we review the basic mechanisms in the pik-calculus, including the message-passing primitives of the pi-calculus, and the notion of atomic failures and logs added by the pik-calculus. In Sect. 3 we describe the representation of causality and causal closure, while in Sect. 4 we describe the mechanisms for atomic commitment. In

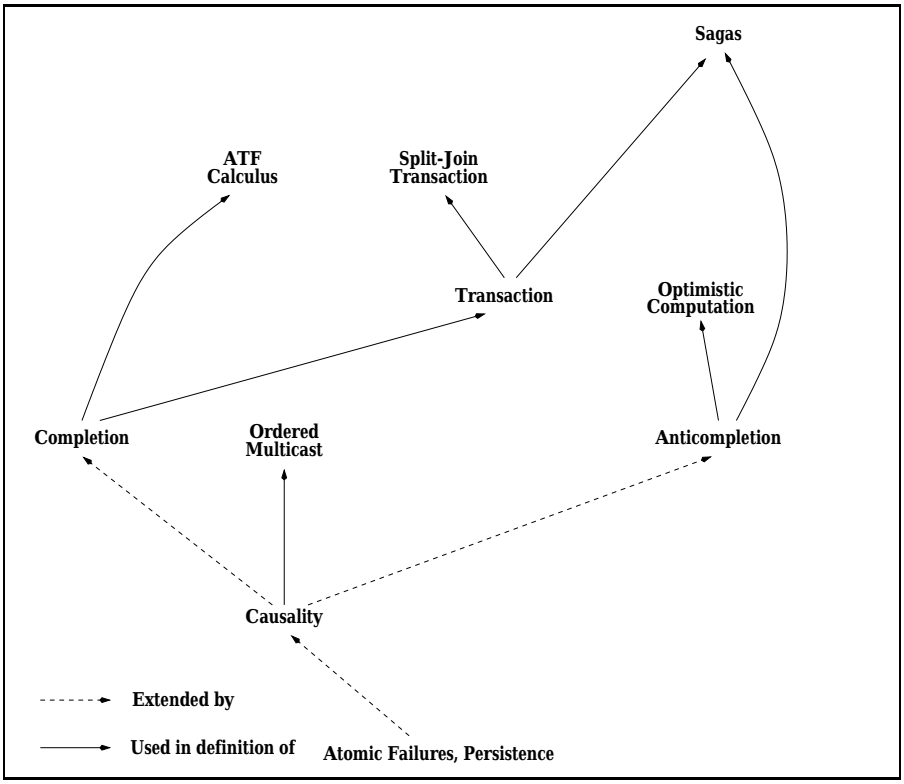


Fig. 1. Applications of Abstractions

Sect. 5 we describe the mechanisms for atomic *anti*commitment, undoing the effects of conclaves that have been committed optimistically. Sect. 6 considers notions of correctness for the calculus. Finally Sect. 7 provides a comparison with related work and conclusions.

2 Atomic Failures and Logs

The syntax for our mini-language, the pik-calculus, is given in Fig. 2. As usual in such calculi, processes are simple "assembly language" concurrent programs, with operations for message-passing. In this case the language is an extension of the asynchronous pi-calculus [36,31,43], a popular calculus for describing distributed programs where channel names are globally unique and may be transmitted in messages. For example a client-server application can be described by having the client send a private reply channel to the server. In addition to constructs for sending and receiving messages, there is also an operation for generating new channel names, for replicating processes (this can be used to define recursive process descriptions) and for forming the parallel composition of processes. For formal reasoning purposes, messages are restricted to tuples of names. For implementation reasons, we further prevent receipt of messages on channels that are not defined locally.

In our extension of this calculus, processes are grouped into collections, called conclaves. A process P executing as part of a conclave c is represented by the network term $c\{P\}$. A network C is a collection of processes executing in conclaves.

$p \in$ Parameter	::=	x	Variable
	|	n, c	Name
$P \in$ Processes	::=	*stop*	Stopped process
	|	*send* $\langle p_1, \ldots, p_k \rangle$ *on* p	Message
	|	*receive* $(x_1, \ldots, x_k)$ *from* n *in* P	Message receive
	|	*new* n *in* P	Scoped name
	|	*repeat* P	Replication
	|	$(P_1 \mid P_2)$	Parallel composition

(a) Core pi-calculus

$P \in$ Processes	::=	*logif* $c\{L\}$ *then* P_1 *else* P_2	Check for log entry
	|	*logawait* $(x_1, \ldots, x_n)c\{L\}$ *in* P	Wait for log entry
	|	*logappend* $\langle \overline{p_k} \rangle$ *with rule-name in* P	Append to log
$C \in$ Networks	::=	*empty*	Empty network
	|	$c\{P\}$	Process in conclave
	|	$c\{L\}$	Log
	|	*new* n *in* C	Scoped name
	|	$(C_1 \mid C_2)$	Parallel composition, wire
$L \in$ Log Entry	::=	*true* $\mid L_1 \wedge L_2$	Empty log, conjunction
	|	$c_1 \rightsquigarrow c_2$	c_1 immediately precedes c_2
	|	*PreClosed* | *Closed*(S) | *Closed*	No further causal predecessors
	|	*PreCommitted* | *Commitable*(c)	Commitment protocol
	|	*Committed* | *Aborted*	Committed, aborted
	|	*Anticommitable*(c, S)	Anticommitment protocol
	|	*UndoAdministrator*(S)	Anticommitment administrator
	|	*Undoable* | *UndoPrepared*(c_1)	Prepared to undo commitment
$S \in$ Name Set	::=	$\{p_1, \ldots, p_k\}$	

(b) Extensions for pik-calculus

Fig. 2. Syntax of the pik-calculus

Besides organizing processes into conclaves, the other innovation in this calculus is the addition of stable storage. Stable or persistent storage in our calculus is represented by a multiset of located propositions, abstractly representing the entries in the "log." Each conclave has a log, represented by a collection of logical propositions L. The fact that a log is for a conclave named c is represented by a "located"

$$empty \mid C \equiv C$$
$$C_1 \mid C_2 \equiv C_2 \mid C_1$$
$$(C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)$$
$$(new\ n\ in\ C_1) \mid C_2 \equiv new\ n\ in\ (C_1 \mid C_2),\ \ n \notin fn(C_2)$$
$$new\ n_1\ in\ new\ n_2\ in\ C \equiv new\ n_2\ in\ new\ n_1\ in\ C$$
$$new\ n\ in\ C \equiv C,\ \ n \notin fn(C)$$
$$true \wedge L \equiv L$$
$$L_1 \wedge L_2 \equiv L_2 \wedge L_1$$
$$(L_1 \wedge L_2) \wedge L_3 \equiv L_1 \wedge (L_2 \wedge L_3)$$
$$c\{stop\} \equiv empty$$
$$c\{P_1 \mid P_2\} \equiv (c\{P_1\} \mid c\{P_2\})$$
$$c\{new\ n\ in\ P\} \equiv new\ n\ in\ c\{P\},\ n \neq c$$
$$stop \mid P \equiv P$$
$$P_1 \mid P_2 \equiv P_2 \mid P_1$$
$$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$
$$(new\ n\ in\ P_1) \mid P_2 \equiv new\ n\ in\ (P_1 \mid P_2),\ \ n \notin fn(P_2)$$
$$new\ n_1\ in\ new\ n_2\ in\ P \equiv new\ n_2\ in\ new\ n_1\ in\ P$$
$$repeat\ P \equiv P \mid repeat\ P$$
$$new\ n\ in\ P \equiv P,\ \ n \notin fn(P)$$

Fig. 3. Equivalence Rules for pik-calculus

log of the form $c\{L\}$. The predicate names for the log propositions are predetermined as part of the calculus, but the calculus could be extended with other forms of log entries.

The semantics for the pik-calculus are specified using various judgement forms:

$P_1 \equiv P_2$	Process equivalence	Fig. 3
$L_1 \equiv L_2$	Log equivalence	Fig. 3
$C_1 \equiv C_2$	Network equivalence	Fig. 3
$C_1 \longrightarrow C_2$	Computation	Fig. 4
$C \models c\{L\}$	Log query	Fig. 4
$C \models c \overset{*}{\rightsquigarrow} c'$	Failure dependency	Fig. 5
$C, (\overline{x_k}) \models \xrightarrow{rule\text{-}name} c\{L\}$	Log append rule	Fig. 5, 6, 9

$$\mathbb{E}[\ \cdot\ ] ::= [\ \cdot\ ] \ \mid\ (\mathbb{E}[\ \cdot\ ] \mid C) \ \mid\ \mathit{new}\ n\ \mathit{in}\ \mathbb{E}[\ \cdot\ ]$$

$$\frac{C_1 = \mathbb{E}[C_1'] \quad C_1' \longrightarrow C_2' \quad C_2 = \mathbb{E}[C_2']}{C_1 \longrightarrow C_2} \qquad \text{(Red Cong)}$$

$$(c_1\{\mathit{send}\ \langle \overline{n_k} \rangle\ \mathit{on}\ n\} \mid c_2\{\mathit{receive}\ (\overline{x_k})\ \mathit{from}\ n\ \mathit{in}\ P\}) \longrightarrow c_2\{\{\overline{n_k}/\overline{x_k}\}P\} \qquad \text{(Red Receive)}$$

$$\frac{c\{L'\} \models c\{\{\overline{n_k}/\overline{x_k}\}L\}}{(c\{L'\} \mid c'\{\mathit{logawait}\ (\overline{x_k})c\{L\}\ \mathit{in}\ P\}) \longrightarrow (c\{L'\} \mid c'\{\{\overline{n_k}/\overline{x_k}\}P\})} \qquad \text{(Red Wait)}$$

$$\frac{c\{L'\} \models c\{L\}}{(c\{L'\} \mid c'\{\mathit{logif}\ c\{L\}\ \mathit{then}\ P_1\ \mathit{else}\ P_2\}) \longrightarrow (c\{L'\} \mid c'\{P_1\})} \qquad \text{(Red IfLogTrue)}$$

$$(c\{L'\} \mid c'\{\mathit{logif}\ c\{L\}\ \mathit{then}\ P_1\ \mathit{else}\ P_2\}) \longrightarrow (c\{L'\} \mid c'\{P_2\}) \qquad \text{(Red IfLogFalse)}$$

$$\frac{C = (C' \mid c\{\mathit{logappend}\ \langle \overline{n_k} \rangle\ \mathit{with\ rule\text{-}name}\ \mathit{in}\ P\} \mid c\{L_1\}) \qquad C, (\overline{x_k}) \models \xrightarrow{\mathit{rule\text{-}name}} c\{L_2\}}{C \longrightarrow (C' \mid c\{P\} \mid c\{L_1 \wedge \{\overline{n_k}/\overline{x_k}\}L_2\})} \qquad \text{(Red Append)}$$

(a) Computation Rules

$$\frac{C_i \models c\{L\} \text{ for some } i \in \{1,2\}}{(C_1 \mid C_2) \models c\{L\}} \qquad \text{(Pred Par)}$$

$$\frac{n \notin \mathit{fn}(L) \cup \{c\} \quad C \models c\{L\}}{(\mathit{new}\ n\ \mathit{in}\ C) \models c\{L\}} \qquad \text{(Pred New)}$$

$$\frac{L \equiv L' \wedge L''}{c\{L\} \models c\{L'\}} \qquad \text{(Pred Log)}$$

(b) Log Query

Fig. 4. Base Semantics of pik-calculus

The structural equivalence rules for processes and conclaves are provided in Fig. 3. The rules for processes P are the usual equivalence rules for the pi-calculus, including extrusion of scope of locally generated names. The rules for conclaves replicate the rules for processes, and also include the following:

$$c\{\mathit{stop}\} \equiv \mathit{empty}$$

$$c\{P_1 \mid P_2\} \equiv c\{P_1\} \mid c\{P_2\}$$

$$c\{\textit{new } n \textit{ in } P\} \equiv \textit{new } n \textit{ in } c\{P\},\ n \neq c$$

These rules relate processes to conclaves; a collection of processes located in conclaves is equivalent to a collection of "atomic" processes, each executing a local operation (message send or message receive, or an operation on the logs as specified below) located in conclaves. Each such atomic process has the form $c\{P\}$ where P has one of the aforesaid forms and where c denotes the "location" of the process. In this calculus, we take processes up to α-conversion (renaming) of scoped names, so that all substitutions over processes and conclaves are assumed to be capture-avoiding.

There are many ways in which conclaves differ from ambients, and we only mention the two that are most relevant. First, ambients do not have the distributivity rule for parallel composition. This reflects the different objectives of the calculi: ambients want to make all communication local, whereas we do not want boundaries for atomic failure to interfere with communication. Therefore we do not complicate our calculus with operations for "navigating" conclaves, whereas such navigation is at the heart of the ambient calculus. Second, conclaves cannot be nested within each other; we do not pursue this complication of the calculus because it is not clear what the motivation for such an extension would be. A desire for such nesting might be motivated by a desire for something analogous to nested transactions [37,35]. However nested transactions are sufficiently complicated in a global computing environment that we prefer to build them up from simpler notions, as alluded to in Sect. 7.

Some operations require examining all of the log entries for a conclave, for example to ensure the absence of a particular log entry. Therefore each conclave c is required to have a single log, of the form $c\{L\}$ where L is a conjunction of log entries. There are three constructs for interacting with stable storage:

$$\begin{aligned} P ::= {} & \textit{logif } c\{L\} \textit{ then } P_1 \textit{ else } P_2 \\ | \ & \textit{logawait } (x_1, \ldots, x_k) c\{L\} \textit{ in } P \\ | \ & \textit{logappend } \langle p_1, \ldots, p_k \rangle \textit{ with rule-name in } P \end{aligned}$$

The semantics for these constructs, as well as the semantics for message-passing, are provided in Fig. 4(a). The *logif* construct allows a process to check for the presence of a particular log entry. There are obvious race conditions with this construct unless it is used carefully. This is reflected in the (RED IFLOGFALSE) rule, that allows the conditional to pessimistically assume that a (presumably remote) log entry is not present. An application using this construct must allow for the fact that this assumption is sometimes wrong. Despite its limitations, this construct is useful in some applications, for example in the translation of the atf-calculus provided in Sect. 4, where it is used to poll a communications channel for a message from a committed process.

The *logawait* construct blocks until log entries matching the pattern are in stable storage. For example an undo action can be specified to have the form

$$\textit{logawait } ()c\{\textit{Aborted}\} \textit{ in } P$$

If the conclave c is aborted, then the process P will execute, presumably undoing some of the effects of the aborted conclave. The transition rule in the operational semantics for the await construct has the form:

$$\frac{c\{L'\} \models c\{\{\overline{n_k}/\overline{x_k}\}L\}}{(c\{L'\} \mid c'\{\mathit{logawait}\ (\overline{x_k})c\{L\}\ \mathit{in}\ P\}) \longrightarrow (c\{L'\} \mid c'\{\{\overline{n_k}/\overline{x_k}\}P\})} \quad \text{(RED WAIT)}$$

This uses the judgement form $C' \models c\{L\}$ to query if the proposition L is present in the logs for the conclave c. The rules for querying the logs are provided in Fig. 4(b). The rules fairly straightforwardly decompose a surrounding context for a log entry of the required form. The general form of the log query rules is useful for checking the preconditions of the log append rules, provided in subsequent sections.

The *logappend* construct is used to add to the contents of stable storage, adding entries in the log. The operations for adding to stable storage are specified by named rules. Each rule requires some preconditions and adds some new collection of propositions to stable storage. The rules are predefined as part of the calculus, in order to ensure some consistency properties of the operational semantics. The These log append rules are specified in Fig. 5, Fig. 6 and Fig. 9, using judgements of the form

$$C, (\overline{x_k}) \models \xrightarrow{\textit{rule-name}} c\{L\}$$

where *rule-name* is the name of the rule, C the surrounding context (used for checking preconditions), c the name of the conclave executing the rule, L the propositions added to storage by the rewrite rule, and $x_1, \ldots, x_k$ are free variables in the pattern given by L. Then the transition rule for changing storage is given by:

$$\frac{\begin{array}{c} C = (C' \mid c\{\mathit{logappend}\ \langle\overline{n_k}\rangle\ \mathit{with\ rule\text{-}name\ in}\ P\} \mid c\{L_1\}) \\ C, (\overline{x_k}) \models \xrightarrow{\textit{rule-name}} c\{L_2\} \end{array}}{C \longrightarrow (C' \mid c\{P\} \mid c\{L_1 \wedge \{\overline{n_k}/\overline{x_k}\}L_2\})} \quad \text{(RED APPEND)}$$

A conclave c can only add to its own log entries. It is possible to check for preconditions in the surrounding context C. This context includes the log for c, $c\{L_1\}$. For some rules it may be necessary for a conclave to examine the logs of other conclaves, though only for positive conditions (the presence, but not the absence, of log entries at other sites). In some cases this may require communication with remote sites holding those logs. These remote sites are captured by the part of the context C'. The semantics abstracts from how communication with remote logs should be done. An obvious approach is to send and receive system messages "under the hood," possibly piggybacked on application messages. This assumes the availability of point-to-point communication channels between processes, which may not always be available. We comment on an alternative approach in the conclusions. The point is that the only remote communication assumptions made in this model is that processes are able to query the log state of remote processes.

Stable storage in our calculus is used to safely save the state of the conclaves, where state is recorded by several proposition types, for causal relationships, commitment protocol state, etc. As a preliminary example, we have specified what it means for a conclave to commit or abort. In a database application, abortion means that updates must be undone and locks released, while commitment means that scheduled updates must be written to the database. We leave the exact semantics of abortion or commitment to the application, and only require that a record be kept in storage of the aborted or committed state of the conclave.

For a conclave that has no causal dependencies with other conclaves, we can supply two rewrite rules, for committing and aborting respectively:

$$\frac{C \not\models c\{\mathit{Aborted}\}}{C, () \models \xrightarrow{\text{CommitEx1}} c\{\mathit{Committed}\}} \qquad \text{(Red Commit Ex1)}$$

$$\frac{C \not\models c\{\mathit{Committed}\}}{C, () \models \xrightarrow{\text{AbortEx1}} c\{\mathit{Aborted}\}} \qquad \text{(Red Abort Ex1)}$$

These rules are an example of the facility of being able to view all log entries for the local conclave when checking the precondition for the rewrite rules, and therefore being able to check the *absence* of certain log propositions. This ability to view all of the log entries relies on the invariant of every conclave having a single log.

For a conclave that has causal predecessors, if all of the causal predecessors of a conclave are committed, then the following rewrite rule allows that conclave to itself commit. The antecedent for the following rule checks that any causal predecessors c' of the conclave c have committed:

$$\frac{C \not\models c\{\mathit{Aborted}\} \quad (C \not\models c\{c' \rightsquigarrow c\} \text{ or } C \models c'\{\mathit{Committed}\}) \text{ for all } c' \in \mathit{fn}(C)}{C, () \models \xrightarrow{\text{CommitEx2}} c\{\mathit{Committed}\}}$$

(Red Commit Ex2)

The log append rules provided in this section are only illustrative examples. The actual rules are provided in the following sections.

3 Causality

Causality is already recognized in the distributed systems community as important for reasoning about distributed computations, in characterizing global states, computing distributed snapshots, designing fault-tolerant replicated systems etc [15,46]. Traditionally, causality is characterized by dependencies induced by messages exchanged between concurrently executing sequential processes (for example, Lamport's "happened-before" relation [33]). However the approach of tracking causal dependencies at the communication level has been criticized [49,14], both for missing dependencies and for detecting "false" dependencies. The former may

$$\frac{C \not\models c\{PreClosed\}}{C,(x) \models \xrightarrow{CausalPred} c\{x \rightsquigarrow c\}} \quad \text{(RED CAUSAL PRED)}$$

$$\frac{C \models x\{c \rightsquigarrow x\}}{C,(x) \models \xrightarrow{CausalSucc} c\{c \rightsquigarrow x\}} \quad \text{(RED CAUSAL SUCC)}$$

$$C,() \models \xrightarrow{PreClosed} c\{PreClosed\} \quad \text{(RED PRECLOSED)}$$

$$\frac{S = \{\overline{c_k}\} = \{c' \in fn(C) \mid C \models c' \overset{*}{\rightsquigarrow} c\} \quad C = (C' \mid \prod \overline{c_k\{L_k\}}) \quad (C \models c'\{PreClosed\} \text{ or } C \models c'\{Closed\}) \text{ for } c' \in S}{C,() \models \xrightarrow{Closed} c\{Closed(S)\}} \quad \text{(RED CLOSED)}$$

(a) Computation rules

$$\frac{C \models c\{c_1 \rightsquigarrow c_2\}}{C \models c_1 \overset{*}{\rightsquigarrow} c_2} \quad \text{(PRED CAUSAL LOG)}$$

$$C \models c \overset{*}{\rightsquigarrow} c \quad \text{(PRED CAUSAL REFL)}$$

$$\frac{C \models c_1 \overset{*}{\rightsquigarrow} c_2 \quad C \models c_2 \overset{*}{\rightsquigarrow} c_3}{C \models c_1 \overset{*}{\rightsquigarrow} c_3} \quad \text{(PRED CAUSAL TRANS)}$$

$$\frac{C \models c\{Closed(S)\}}{C \models c\{Closed\}} \quad \text{(PRED CLOSED)}$$

(b) Log query rules

Fig. 5. Semantics of Causality in the pik-calculus

happen because of hidden channels outside the communication system (for example, physical pressure in a pipe), while the latter may happen because there is no causal dependency (at the application level) between a message that is sent and a message that was received just before the message send. Cheriton and Skeen [14] argue that what is required is a mechanism for tracking causal dependencies at the application level rather than the communication level, since the application can be aware of hidden channels and can avoid false dependencies.

Given the link between atomic failure and causality established by causal consistency, we use conclaves as a mechanism for tracking causality at the application level. Causality is not a relationship between message send and receive events, but rather is a failure dependency relationship between conclaves: if a conclave fails, then conclaves that depend on it are also required to fail. Furthermore, it would be a mistake to track causal dependencies based on communication between con-

claves. For example two conclaves on different machines may communicate via firewall daemons, but we would not expect a firewall daemon to fail because a process whose message it delivered failed. Instead we allow the application itself to assert causal dependencies between conclaves. Because of this, there is no scalable way to prevent cycles in the causal dependency graph.

This has implications for completing distributed conclaves. To maintain causal consistency, a conclave cannot commit until all of its causal predecessors have committed or are also willing to commit. Because of causal cycles, it may be necessary for several conclaves (all of the members of a strongly connected component in the causality graph) to commit simultaneously. We require each conclave to execute at a specific network site, but conclaves may communicate with other conclaves at other sites. So it may be necessary to run an atomic commitment protocol involving several conclaves over an unreliable network. Since this is in general an unsolvable problem, we adopt an approach that can be the basis for widely-used protocols such as two-phase commit and early-prepare commit, but it is not tied to any particular protocol.

When a transaction aborts, its effects must be undone. If changes have been made to a database, the previous values of the changed variables must be restored. We do not provide automatic support for undoing the effects of conclaves that abort. Except for the special case of databases, it is not clear what form undoing should take. For example, undoing a transfer of funds may involve sending an attorney's letter through the ordinary mail. In any case if the receiver of the original message has accepted a causal dependency on the sender of the message, the receiver will be prevented from completing.

Causality is recorded in storage using located propositions of the form $c\{c' \rightsquigarrow c\}$, recording (in c's logs) that c' is a causal predecessor of c. These propositions are added by processes using the *logappend* construct, using the (RED CAUSAL PRED) transition rule in Fig. 5. Causal predecessors may also have log entries of the form $c'\{c' \rightsquigarrow c\}$, recording their successors. To ensure mutual consistency we require that a log entry recording a causal successor be justified by the presence of a log entry recording a causal predecessor. This is enforced by the antecedent of the (RED CAUSAL SUCC) rule for adding a log entry recording a causal successor. The causal succcessor log entries are required for anticommitment, as explained in Sect. 5. The causal log entries give rise to a reflexive transitive relation $c_1 \overset{*}{\rightsquigarrow} c_2$, with rules given in Fig. 5.

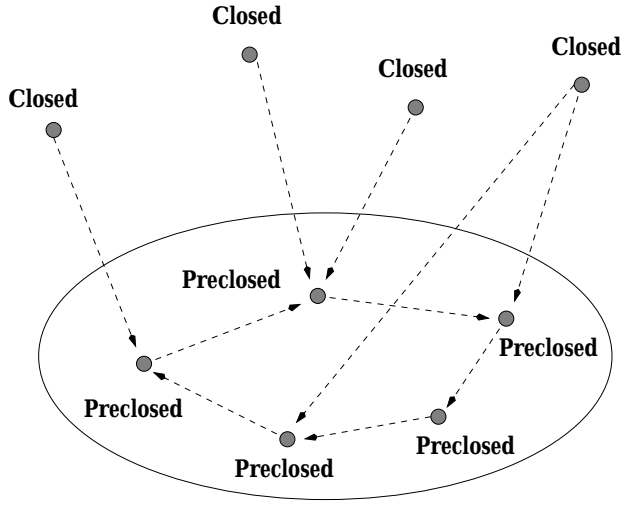
Some decisions must be made based on the assumption that all causal predecessors of a conclave are known, i.e., causal predecessors cannot be added after such decisions are made. Therefore we add another log entry type $c\{\mathit{Closed}(S)\}$ that denotes that no further causal predecessors can be added for the conclave c, and that S (a set of conclave names) is the set of all causal predecessors of c. We also add another proposition *Closed* that is derived from the former proposition type, and simply denotes that the set of causal predecessors is closed.

A conclave cannot autonomously close up the set of its causal predecessors, because one of its predecessors may be open to causal extensions. A conclave could

wait until each of its predecessors are closed before it closes itself up to further extensions. However this may lead to deadlock if there are causal cycles. We therefore add a buffer state, *PreClosed*, that a conclave can transition to unconditionally. A transition to the *Closed* state is enabled when all of the causal predecessors of a conclave, including the conclave itself, are in the *PreClosed* state. This is given by the (RED CLOSED) transition in Fig. 5.

In the latter transition, the set S is the set of all causal predecessors, inverse transitively closed, of the conclave c. The transition rule requires that the logs for all of these predecessors are in the context of the log append operation. This is in order to check that all of the predecessors are causally closed or prepared to be closed. If this condition is met, the conclave c becomes causally closed.

Other protocols for causal closure can be built on top of this framework, using the transitivity property of causality to trim the logs. For example, a conclave must have evidence that all of its causal predecessors, not just its immediate causal predecessors, are either in the closed or preclosed state. Using the property that if a conclave is causally closed, then all of its causal predecessors are closed, a protocol can restrict its attention to all of the conclaves in a strongly connected component of the causality graph. A coordinator for the strongly connected component can use a two-phase commit protocol to check that all conclaves in the component are preclosed, and that all immediate causal predecessors outside the component are closed, to authorize the transition of the conclaves in the component to the closed state:



In this picture, nodes represent conclaves in various states (causally closed, prepared to close), edges represent causal dependencies, and the large oval represents a collection of mutually dependent conclaves that can cooperate using some atomic commitment protocol to achieve causal closure, using the fact that predecessor conclaves outside this collection are already closed.

$$\frac{C \not\models c\{PreClosed\}}{C, () \models \xrightarrow{AtStUndo} c\{Undoable\}} \quad \text{(Red At StUndo)}$$

$$\frac{C \not\models c\{L\} \text{ for } L \in \{PreCommitted, Committed, UndoPrepared(c'), UndoAdministrator(S)\}}{C, () \models \xrightarrow{AtStAbort} c\{Aborted\}} \quad \text{(Red At StAbort)}$$

$$\frac{C \not\models c\{L\} \text{ for } L \in \{Aborted, UndoPrepared(c'), UndoAdministrator(S)\}}{C, () \models \xrightarrow{AtStPreCommit} c\{PreCommitted\}} \quad \text{(Red At StPreCmt)}$$

$$\frac{C \models c' \overset{*}{\rightsquigarrow} c \quad C \models c'\{Aborted\}}{C, () \models \xrightarrow{AtPcAbort} c\{Aborted\}} \quad \text{(Red At PcAbort)}$$

$$\frac{\begin{array}{c} C \models c\{PreCommitted\} \quad C \models c\{Closed(S)\} \\ C \not\models c\{L\} \text{ for } L \in \{UndoPrepared(c'), UndoAdministrator(S')\} \\ \big((C \models c'\{Closed\} \text{ and } C \models c'\{PreCommitted\}) \text{ or } C \models c'\{Committed\}\big) \text{ for } c' \in S \\ (C \not\models c\{c' \rightsquigarrow c\} \text{ or } C \models c'\{Commitable(c)\}) \text{ for } c' \in S \end{array}}{C, () \models \xrightarrow{AtPcCommit} c\{Committed\}} \quad \text{(Red At PcCommit)}$$

(a) Computation rules

$$\frac{C \models c'\{PreClosed\} \text{ and } (C \not\models c'\{Undoable\} \text{ or } C \models c'\{c' \rightsquigarrow c\})}{C \models c'\{Commitable(c)\}} \quad \text{(Pred At PcCommit)}$$

(b) Log query rules

Fig. 6. Semantics of Commitment in the pik-calculus

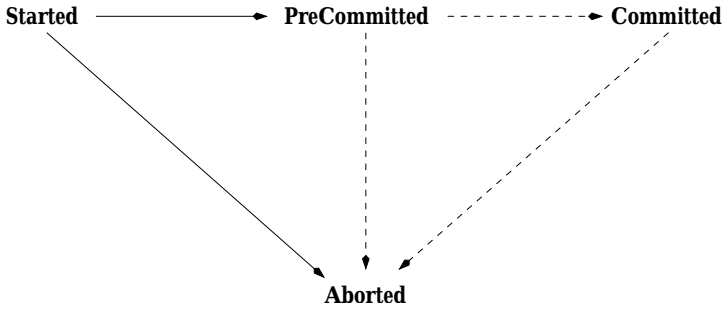
4 Commitment

A conclave encapsulates a set of processes that perform some set of actions that eventually either succeed or fail. We refer to these alternatives as *commitment* and *abortion*. Commitment builds on causality, since a conclave cannot commit unless all of its causal predecessors commit. It is tempting to define a transition rule that allows a transition to the *Committed* state if all of the causal predecessors of the conclave are in the *Committed* state. However (as with causal closure) this is insufficient if there are causal cycles. As with causal closure, we introduce a buffer state *PreCommitted*, to which a conclave in the started state can transition uncondition-

ally (Rule (RED AT STPRECMT) in Fig. 6). The transition to the *Committed* state is enabled when all causal predecessors are either committed, or are precommitted and are causally closed (Rule (RED AT PCCOMMIT)).

The last line of the antecedents for the (RED AT PCCOMMIT) rule refers to anti-commitment, described in more detail in the next section. If a conclave is undoable (it can subsequently be uncommitted), then the rule for committing requires that its immediate causal predecessors be recorded in the logs (all other conclaves are only required to record immediate causal successors, from which the causality relation is derived). The aforesaid last line of the antecedents requires that if c' is an immediate causal predecessor of c, then c must be commitable with respect to c' (represented by the derived log entry $Commitable(c)$ at c'). The latter log entry, derived using the (PRED AT PCCOMMIT) rule, states that the conclave c' is causally closed and has a log entry recording c as an immediate successor if c' is undoable. Checking the closure condition on c' at the same time as the undoability condition ensures that c' cannot subsequently be made undoable.

In contrast with causal closure, there is also the possibility of transitioning to an *Aborted* state. A conclave that has not yet precommitted can abort unconditionally (Rule (RED AT STABORT)). A conclave in the precommitted state can only abort if one of its causal predecessors has aborted (Rule RED AT PCABORT). A summary of the possible state transitions for a conclave is given by:



The transition from the committed state to the aborted state is provided by anti-commitment, described in Sect. 5.

Example: Distributed Transactions.

Ignoring aspects of locking for now, we identify a distributed transaction [5] with a collection of conclaves. There is a parent conclave for the start of the transaction at the original site. This transaction invokes operations at remote sites; each instance of the transaction at a remote site is represented by a conclave at that remote site, spawned by the parent conclave. Each child conclave accepts a causal dependency on the parent conclave, and vice versa. At the conclusion of the transaction, the parent conclave executes a two-phase commit protocol:

(i) The parent *conclave* acts as an administrator for the protocol. It contacts each child conclave to induce the latter to enter the *PreCommitted* state, and determines if any children have failed.

(ii) If all children have entered the *PreCommitted* state, "evidence" of this is gath-

ered by the administrator and transmitted to the children. The latter use this evidence to enter the *Committed* state. Otherwise the administrator aborts and induces the remaining children to abort (enter the *Aborted* state).

Example: Split-Join Transactions.

Such transactions support a split operation that breaks a transaction into two transactions, and a join operation that does the converse [40]. We can define a *split* operation if we extend the calculus with the following operation for "forking" code that executes within another conclave, as well as an operation for initializing the log of a new conclave:

$$\begin{aligned} P \in \text{Process} ::= {} & c\{P\} \\ \mid {} & \mathit{loginit\ in\ } P \end{aligned}$$

with the operational rules

$$c_1\{c_2\{P\}\} \longrightarrow c_2\{P\} \qquad \text{(Red Exec)}$$

$$\frac{C = \prod \overline{c_k\{P_k\}} \quad c \notin \{\overline{c_k}\}}{\mathit{new\ c\ in\ } (c\{\mathit{loginit\ in\ } P\} \mid C) \longrightarrow \mathit{new\ c\ in\ } (c\{\mathit{true}\} \mid c\{P\} \mid C)} \qquad \text{(Red LogInit)}$$

The forking construct $c\{P\}$ still does not imply any nesting of conclaves, rather it is simply a way for a conclave to extrude code that executes in another conclave. The *loginit* construct allows an initial empty log to be created for a conclave. The syntactic restriction on the context C, and the static scoping rules for the pi-calculus, ensure that there are no other logs for this conclave present. Then define

$$(\mathit{split\ c\ in\ } P_2 \mathit{\ then\ } P_1) \equiv (\mathit{new\ c\ in\ } (c\{\mathit{loginit\ in\ } P_2\} \mid P_1))$$

Then

$$\begin{aligned} c_1\{\mathit{split\ } c_2 \mathit{\ in\ } P_2 \mathit{\ then\ } P_1\} &\equiv c_1\{\mathit{new\ } c_2 \mathit{\ in\ } c_2\{\mathit{loginit\ in\ } P_2\} \mid P_1\} \\ &\equiv \mathit{new\ } c_2 \mathit{\ in\ } c_1\{c_2\{\mathit{loginit\ in\ } P_2\} \mid P_1\} \\ &\equiv \mathit{new\ } c_2 \mathit{\ in\ } (c_1\{c_2\{\mathit{loginit\ in\ } P_2\}\} \mid c_1\{P_1\}) \\ &\longrightarrow \mathit{new\ } c_2 \mathit{\ in\ } (c_2\{\mathit{loginit\ in\ } P_2\} \mid c_1\{P_1\}) \\ &\longrightarrow \mathit{new\ } c_2 \mathit{\ in\ } (c_2\{\mathit{true}\} \mid c_2\{P_2\} \mid c_1\{P_1\}). \end{aligned}$$

The join operation is provided by having the conclaves for the two transactions become mutually dependent on each other.

Example: ATF Calculus.

The atf-calculus [18] is a process calculus with atomic failure as its central organizing principle, with a guarantee of a failure-free execution trace being derivable from any execution trace (based on ignoring the effects of failed processes). This example demonstrates how particular patterns of programming with atomic failures can be constructed atop the primitives of the pik-calculus.

The syntax of the atf-calculus is given by:

$$
\begin{aligned}
A \in \mathit{Process} ::= {} & \mathit{stop} \mid \mathit{repeat}\ A \mid (A_1 \mid A_2) \mid \mathit{new}\ x\ \mathit{in}\ A \\
\mid {} & \mathit{send}\ \langle \overline{p_k} \rangle\ \mathit{on}\ p \mid \mathit{receive}\ (\overline{x_k})\ \mathit{from}\ n\ \mathit{in}\ A \\
\mid {} & \mathit{receive\ committed}\ (\overline{x_k})\ \mathit{from}\ n\ \mathit{in}\ A \\
\mid {} & \mathit{prepare} \mid \mathit{abort} \mid \mathit{commit} \\
T \in \mathit{Transaction} ::= {} & \mathit{empty} \mid t\{A\} \mid t\{L\} \mid (T_1 \mid T_2) \mid \mathit{new}\ n\ \mathit{in}\ T \\
L \in \mathit{Log} ::= {} & \mathit{true} \mid L_1 \wedge L_2 \mid t\{\mathit{prepare}\} \mid t\{\mathit{abort}\} \mid t\{\mathit{commit}\} \\
\mid {} & (t_1\{\mathit{send}\ \langle \overline{n_k} \rangle\ \mathit{on}\ n\} \mid t_2\{\mathit{receive}\ (\overline{x_k})\ \mathit{from}\ n\ \mathit{in}\ A\})
\end{aligned}
$$

The semantics of the atf-calculus are presented in Fig. 7.

Besides the usual operations for stopped processes, replication, parallel composition, new name generation, and message passing, there are operations for aborting and committing transactions. Commitment is similar to the *early prepare commit protocol* [48], whereby participants prepare for commitment and an administrator then decides if the participants should commit. It is also similar to completion in the pik-calculus, except that the latter has more freedom for the application to decide when to enter the phases of the commitment protocol. The *prepare* operation puts a participant into the prepared state, recorded by a log entry of the form $\mathcal{L}(t\{\mathit{prepare}\})$. A participant can commit if all of its causal predecessors are either committed or prepared. A participant can abort if any of its causal predecessors has aborted. A participant can autonomously abort any time before it enters the prepared state.

As with the pik-calculus, there are logs of the form $t\{L\}$, at most one per transaction. There are four log entry types, of the form $t\{A\}$. Three of these log entry types record if a transaction is in the prepared, committed or aborted state. The fourth log entry type records that (a process in) the transaction t_2 received a message sent by (a process in) the transaction t_1. This log entry type implicitly records a causal dependency from t_1 to t_2; it is also useful for undoing the effects of an aborted transaction, retransmitting any messages that the transaction had received before aborting. This is done by the (RED ATF UNDO) rule in Fig. 7.

There are two operations for receiving messages. The first operation corresponds to receiving a message and accepting a causal dependency on the sending transaction. A transaction cannot in general receive new messages if it has entered the prepared state, since that might introduce new causal dependencies on transactions that could then abort, invalidating any earlier decision to commit. This is enforced by the antecedent in the (RED ATF RECEIVE) rule. However the second message receive operation restricts received messages to those that were sent by committed transactions (so a process can isolate itself from the effects of uncommitted transactions). Once a transaction has entered the prepared state, it can only receive messages from other committed transactions. This prevents further causal dependencies from being introduced while committing a transaction, and prevents a committed transaction from gaining a causal dependency on an aborted transaction. This is enforced by the (RED ATF RECVCOMM) rule for receiving only messages from committed transactions. For the first form of the message receive operation, a log is kept in the receiving transaction, both to record causal dependencies and also

to allow message receives to be undone if the receiver subsequently aborts. We do not record the second form of the message receive operation, since such events do not affect commitability and are never undone (the sender will never abort).

The remaining rules are reasonably straightforward. The (Red ATF Abort) and (Red ATF Prepare) rules allow a transaction to abort or enter the prepare state, respectively, provided it has not yet entered any other state. The (Red ATF PrepAbort) rule allows a transaction to abort while it is in the prepare state, if one of its causal predecessors has aborted. The (Red ATF Commit) rule allows a transaction to commit, provided all of its causal predecessors are either in the prepared state or have committed. Finally the (Red ATF Undo) rule allows a message receive to be undone, using the logs, once the receiving process has aborted.

The translation of the atf-calculus into the pik-calculus is provided in Fig. 4. This translation is specified using various metafunctions:

$\mathcal{T}[\![T]\!]\eta$	Translation of transactions to conclaves
$\mathcal{T}[\![L]\!]\eta$	Translation of log entries to undo code
$\mathcal{L}[\![L]\!]S$	Translation of atf-calculus logs to pik-calculus logs
$\mathcal{P}[\![A]\!]t$	Translation of atf-calculus processes to pik-calculus processes

The translation of transactions $\mathcal{T}[\![T]\!]\eta$ is parameterized by a mapping η from each currently visible transaction t to the set of its of immediate causal predecessors. This is used to compute the set of causal predecessors S when the translation generates log entries of the form $\mathit{Closed}(S)$.

For a log L, the translation $\mathcal{T}[\![L]\!]\eta$ constructs a collection of processes process that wait for the corresponding transaction to abort and then resend messages that were received by that transaction. The translation $\mathcal{L}[\![L]\!]S$ converts from atf-calculus log entries to pik-calculus log entries. The top-level translation $\mathcal{T}[\![T]\!]\eta$ invokes both of the aforesaid translations when applied to a log in the atf-calculus; otherwise the only other interesting case is for processes, where it invokes the translation $\mathcal{P}[\![A]\!]c$ translating from atf-calculus processes to pik-calculus processes.

In the latter translation, in the translation of the first message receive operation, messages are augmented with the conclave identifier of the transaction sending the message, and this is used at the receiver to record the causal dependency between sender and receiver. For the second message receive operation, the *receive-commit* operation that only receives messages from committed transactions, the receiving operation polls the input channel until it receives a message that was sent by a transaction that has committed. This is where we make use of the *logif* construct, and demonstrates its usefulness even though the construct may nondeterministically pick the false branch even when the log entry is present: the process in this case simply loops to check the condition again, for another message on the channel. The process that does this polling is defined by the recursive metafunction *RECVCOMM*; this can be translated into a non-recursive process description in the usual manner [36].

5 Anticommitment

Our calculus provides mechanisms for *anticommitment*. A problem with transactions is their unsuitability for long-lived applications [19], since they retain locks on database variables until the transaction eventually commits. One solution to

$$\frac{T_1 = t_1\{\mathit{send}\ \langle n_1,\ldots,n_k\rangle\ \mathit{on}\ n\} \quad T_2 = t_2\{\mathit{receive}\ (x_1,\ldots,x_k)\ \mathit{from}\ n\ \mathit{in}\ P\} \quad t_2\{L\} \not\models t_2\{\mathit{prepare}\}}{T_1 \mid T_2 \mid t_2\{L\} \longrightarrow t_2\{L \wedge (T_1 \mid T_2)\} \mid t_2\{\{\overline{n_k}/\overline{x_k}\}P\}}$$

(RED ATF RECEIVE)

$$\frac{T_1 = t_1\{\mathit{send}\ \langle n_1,\ldots,n_k\rangle\ \mathit{on}\ n\} \quad T_2 = t_2\{\mathit{receive\ committed}\ (x_1,\ldots,x_k)\ \mathit{from}\ n\ \mathit{in}\ P\} \quad t_1\{L\} \models t_1\{\mathit{commit}\}}{T_1 \mid T_2 \mid t_1\{L\} \longrightarrow t_1\{L\} \mid t_2\{\{\overline{n_k}/\overline{x_k}\}P\}}$$

(RED ATF RECVCOMM)

$$\frac{t\{L\} \not\models t\{\mathit{prepare}\}, t\{\mathit{commit}\}}{t\{\mathit{abort}\} \mid t\{L\} \longrightarrow t\{L \wedge t\{\mathit{abort}\}\}}$$

(RED ATF ABORT)

$$\frac{t\{L\} \not\models t\{\mathit{abort}\}}{t\{\mathit{prepare}\} \mid t\{L\} \longrightarrow t\{L \wedge t\{\mathit{prepare}\}\}}$$

(RED ATF PREPARE)

$$\frac{t\{L\} \not\models t\{\mathit{commit}\} \quad T \mid t\{L\} \models t\{\mathit{prepare}\},\ t'\{\mathit{abort}\},\ t' \overset{*}{\rightsquigarrow} t}{t\{\mathit{abort}\} \mid t\{L\} \mid T \longrightarrow t\{L \wedge t\{\mathit{abort}\}\} \mid T}$$

(RED ATF PREPABORT)

$$\frac{S = \{\overline{t_k}\} = \{t' \in fn(T) \mid T \models t' \overset{*}{\rightsquigarrow} t\} \quad T' = \prod \overline{t_k\{L_k\}} \quad T = T' \mid t\{L\} \quad T \models t\{\mathit{prepare}\} \quad T \models t'\{\mathit{commit}\} \text{ or } T \models t'\{\mathit{prepare}\} \text{ for } t' \in S}{t\{\mathit{commit}\} \mid t\{L\} \mid T' \longrightarrow t\{L \wedge t\{\mathit{commit}\}\} \mid T'}$$

(RED ATF COMMIT)

$$\frac{T_1 = t_1\{\mathit{send}\ \langle n_1,\ldots,n_k\rangle\ \mathit{on}\ n\} \quad T_2 = t_2\{\mathit{receive}\ (x_1,\ldots,x_k)\ \mathit{from}\ n\ \mathit{in}\ P\}}{t_2\{L \wedge (T_1 \mid T_2) \wedge t_2\{\mathit{abort}\}\} \longrightarrow T_1 \mid t_2\{L \wedge t_2\{\mathit{abort}\}\}}$$

(RED ATF UNDO)

(a) Computation rules

$$\frac{T_1 = t_1\{\mathit{send}\ \langle n_1,\ldots,n_k\rangle\ \mathit{on}\ n\} \quad T_2 = t_2\{\mathit{receive}\ (x_1,\ldots,x_k)\ \mathit{from}\ n\ \mathit{in}\ P\}}{t_2\{L \wedge (T_1 \mid T_2)\} \models t_1 \overset{*}{\rightsquigarrow} t_2}$$

(PRED ATF CAUSAL HYPOTH)

(b) Log query rules

Fig. 7. Semantics of ATF Calculus

$$\mathcal{T}[\![new\ t\ in\ T]\!]\eta = new\ t\ in\ \mathcal{T}[\![T]\!](\eta[t \mapsto S]) \text{ where } T \equiv (new\ \overline{n}\ in\ (t\{L\} \mid \overline{T}) \text{ and } S = \{t' \mid t\{L\} \models t' \overset{*}{\rightsquigarrow} t\}$$

$$\mathcal{T}[\![t\{L\}]\!]\eta = \mathcal{T}[\![L]\!]\eta \mid t\{\mathcal{L}[\![L]\!]S\} \text{ where } S = F_\infty(t,\eta)$$

$$F_0(t,\eta) = \{t\} \quad F_{k+1}(t,\eta) = \bigcup\{F_k(t',\eta) \mid t' \in \eta(t)\} \quad F_\infty(t,\eta) = \bigcup_{k=0}^{\infty} F_k(t,\eta)$$

$$\mathcal{T}[\![new\ n\ in\ T]\!]\eta = new\ n\ in\ \mathcal{T}[\![T]\!]\eta \qquad \mathcal{T}[\![T_1 \mid T_2]\!]\eta = (\mathcal{T}[\![T_1]\!]\eta \mid \mathcal{T}[\![T_2]\!]\eta) \qquad \mathcal{T}[\![t\{A\}]\!]\eta = t\{\mathcal{P}[\![A]\!]t\}$$

$$\mathcal{T}[\![L]\!]\eta = \begin{cases} t_2\{logawait\ t_1\{Aborted\}\ in\ send\ \langle t_1, \overline{n_k}\rangle\ on\ n\} & \text{if } L = \mathcal{L}(T_1 \mid T_2) \\ & \text{and } T_1 = t_1\{send\ \langle \overline{n_k}\rangle\ on\ n\} \text{ and } T_2 = t_2\{receive\ (\overline{x_k})\ from\ n\ in\ A'\} \\ empty & \text{if } L \in \{t\{prepare\}, t\{commit\}, t\{abort\}, true\} \end{cases}$$

$$\mathcal{T}[\![L_1 \wedge L_2]\!]\eta = (\mathcal{T}[\![L_1]\!]\eta \mid \mathcal{T}[\![L_2]\!]\eta)$$

$$\mathcal{L}[\![L]\!]S = \begin{cases} PreCommitted \wedge PreClosed & if\ L = t\{prepare\} \\ Committed \wedge Closed(S) & \text{if } L = t\{commit\} \\ Aborted & if\ L = t\{abort\} \\ t_2 \rightsquigarrow t_1 & \text{if } L = T_1 \mid T_2 \\ & \text{and } T_1 = t_1\{send\ \langle \overline{n_k}\rangle\ on\ n\} \text{ and } T_2 = t_2\{receive\ (\overline{x_k})\ from\ n\ in\ A'\} \end{cases}$$

$$\mathcal{L}[\![L_1 \wedge L_2]\!]S = (\mathcal{L}[\![L_1]\!]S \wedge \mathcal{L}[\![L_2]\!]S)$$

$$\mathcal{P}[\![A]\!]t = logif\ t\{Committed\}\ then\ (new\ n'\ in\ receive\ ()\ from\ n'\ in\ stop)\ else\ P$$
if $A = receive\ (\overline{x_k})\ from\ n\ in\ A'$
where $P = receive\ (y, \overline{x_k})\ from\ n\ in\ ((logappend\ \langle y\rangle\ with\ CausalPred\ in\ \mathcal{P}[\![A']\!]t) \mid (logawait\ t\{Aborted\}\ in\ send\ \langle y, \overline{x_k}\rangle\ on\ n))$

$$\mathcal{P}[\![A]\!]t = RECVCOMM(n, \mathcal{P}[\![A']\!]t, t) \text{ if } A = receive\ committed\ (\overline{x_k})\ from\ n\ in\ A'$$
where $RECVCOMM(n,P,t) = receive\ (y, \overline{x_k})\ from\ n\ in\ logif\ y\{Committed\}\ then\ P_1\ else\ P_2$
and $P_1 = ((logappend\ \langle y\rangle\ with\ CausalPred\ in\ \mathcal{P}[\![A']\!]t) \mid (logawait\ t\{Aborted\}\ in\ send\ \langle y, \overline{x_k}\rangle\ on\ n))$
and $P_2 = (send\ \langle y, \overline{x_k}\rangle\ on\ n \mid REVCOMM(n,P,t))$

$$\mathcal{P}[\![new\ n\ in\ A]\!]t = new\ n\ in\ \mathcal{P}[\![A]\!]t \qquad \mathcal{P}[\![A_1 \mid A_2]\!]t = (\mathcal{P}[\![A_1]\!]t \mid \mathcal{P}[\![A_2]\!]t) \qquad \mathcal{P}[\![send\ \langle \overline{p_k}\rangle\ on\ p]\!]t = send\ \langle t, \overline{p_k}\rangle\ on\ p$$

Fig. 8. Translation of the atf-calculus

$$\frac{C \models c\{L\} \text{ for } L \in \{PreCommitted, Committed\} \quad C \models c\{Undoable\} \quad C \not\models c\{L\} \text{ for } L \in \{UndoPrepared(c'), UndoAdministrator(S')\}}{C, (x_1, \ldots, x_k) \models \xrightarrow{AntiAdmin} c\{UndoAdministrator(\{c, x_1, \ldots, x_k\})\}} \text{(Red Anti Admin)}$$

$$\frac{C \models c\{L\} \text{ for } L \in \{PreCommitted, Committed\} \quad C \models c\{Undoable\} \quad C \not\models c\{L\} \text{ for } L \in \{UndoPrepared(c'), UndoAdministrator(S')\}}{C, (x) \models \xrightarrow{AntiPrep} c\{UndoPrepared(x)\}} \text{(Red Anti Prep)}$$

$$\frac{C \models c\{UndoAdministrator(S)\} \quad S = \{\overline{c_k}\} \quad C = (C' \mid \prod \overline{c_k\{L_k\}}) \quad C \models c_i\{Anticommitable(c, S_i)\},\ i = 1, \ldots, k \quad C \models c'\{Aborted\}) \text{ for all } c' \in \bigcup_k S_k - S}{C, () \models \xrightarrow{AntiAdmCommit} c\{Aborted\}} \text{(Red Anti AdmAbort)}$$

$$\frac{C \models c\{UndoPrepared(c_0)\} \quad C \models c_0\{Aborted\}}{C, () \models \xrightarrow{AntiPartCommit} c\{Aborted\}} \text{(Red Anti PartAbort)}$$

(a) Computation rules

$$\frac{C \models c'\{c' \rightsquigarrow c''\} \text{ for } c'' \in S \quad C \not\models c'\{c' \rightsquigarrow c''\} \text{ for } c'' \notin S \quad C \models c'\{UndoPrepared(c)\}}{C \models c'\{Anticommitable(c, S)\}} \text{(Pred Anti Comm)}$$

(b) Log query rules

Fig. 9. Semantics of Anticommitment in the pik-calculus

this problem is to optimistically commit, making effects visible, and then provide a mechanism for subsequently undoing the commitment if necessary. We provide the latter mechanism through support for atomic anticommitment, atomically transforming a collection of committed conclaves to aborted conclaves. The causal consistency restriction requires that all causal successors of the anticommitted conclaves be aborted before the conclaves are anticommitted. Anticommitment, in combination with causality, constitutes support for optimistic computation in our language.

Commitment allows a collection of mutually dependent conclaves to commit, provided the result is causally consistent: there is no committed conclave that has aborted causal predecessors. Anticommitment is useful if, for practical purposes, a collection of conclaves optimistically commits "early" and must subsequently be aborted. This constitutes the support for *optimistic computation* in our calculus.

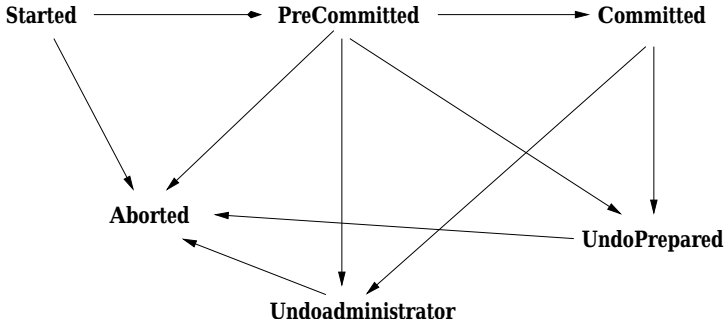
Anticommitment allows a collection of committed conclaves to abort, provided again that the result is causally consistent: there is no aborted conclave that has committed causal successors.

To enable the consistency check, a conclave that can be "anticommitted" must have asserted itself to be undoable (added the *Undoable* proposition to the logs) before committing. As noted in the previous section, the rule for committing then requires that all undoable conclaves must have recorded their immediate causal predecessors in the logs.

With commitment, conclaves transition to the precommitted state. From there they can only transition to the aborted or committed state, in the former case only if a causal predecessor has aborted. It is safe for a conclave to transition to the committed state once all of its causal predecessors have either committed or precommitted, since the precommitted predecessors must then eventually commit. In contrast, if we are to allow anticommitment to fail (i.e., it is not possible to abort a collection of conclaves), it is possible for a conclave that has prepared to anticommit and abort to revert back to the committed state. This leads to a potential race condition with other conclaves that have chosen to anticommit based on the readiness of this conclave to anticommit.

For the current extended abstract, we omit a description of how a conclave that is prepared to anticommit may revert back to the committed state. However we give a formulation of anticommitment that avoids the aforesaid race condition, if we do allow such reversion. Anticommitment is based on a two-phase commit protocol, where a collection of conclaves that desire to anticommit choose (using some application-level protocol) some conclave to be the administrator for the protocol. Once the participants have entered the *UndoPrepared*(c) state, where c is the name of the administrator conclave, they cannot leave this state until the administator has transitioned to the aborted state. On the other hand, the administrator cannot transition to the aborted state until all participants have transitioned to the *UndoPrepared*(c) state.

So the possible states that a conclave can be in are provided by the following:



The rules for anticommitment are provided in Fig. 9. The (RED ANTI ADMIN) and (RED ANTI PREP) rules allow a conclave that is precommitted or committed to become an administrator or a participant, respectively, in an anticommitment protocol. The (RED ANTI ADMABORT) rule allows the administrator of such a protocol to abort, provided that all other conclaves in the protocol have become participants,

$$\frac{C_1 \textbf{ logged } S_1 \quad C_2 \textbf{ logged } S_2 \quad S_1 \cap S_2 = \{\}}{(C_1 \mid C_2) \textbf{ logged } S_1 \cup S_2} \qquad \text{(CONS PAR)}$$

$$\frac{C \textbf{ logged } S}{(new\ n\ in\ C) \textbf{ logged } S - \{n\}} \qquad \text{(CONS NEW)}$$

$$c\{L\} \textbf{ logged } \{c\} \qquad \text{(CONS CONJ)}$$

$$\frac{C \in \{c\{P\}, empty\}}{C \textbf{ logged } \{\}} \qquad \text{(CONS NOTLOG)}$$

Fig. 10. Well-formedness Rules for pik-calculus

and all causal successors of the administrator and the participants that are not included have aborted. This rule requires that the log entries for all of the participants in the anticommitment protocol be present in the context, and the anticommitability predicate is used to obtain the immediate causal successors of the participants in the protocol. The (RED ANTI PARTABORT) rule allows a participant to abort once the administrator has decided to abort.

Example: Sagas.

A saga [25,24] is a collection of transactions $T_1, \ldots, T_k$ that execute in sequence. If transaction T_i aborts, for $i \in \{1, \ldots, k\}$, then none of the subsequent transactions execute, and moreover a collection of "antitransactions" [32] $T_{k-1}^{-1}, \ldots, T_1^{-1}$ execute in sequence. So the end of a run of a saga is either $T_1, \ldots, T_k$ or $T_1, \ldots, T_i, T_i^{-1}, \ldots, T_1^{-1}$.

Sagas are implemented fairly obviously using anticommitment. In addition anticommitment generalizes the approach of sagas to allow arbitrary dependency graphs, not just the simple linear ordering provided with sagas.

6 Correctness

We have two notions of well-formedness for processes; one is a simple syntactic condition that every conclave have no more than one log, while the other is a more sophisticated condition on the consistency of the logs. The latter consistency rules check that certain antecedents hold in the logs if a particular form of log entry is present. For example, if a conclave has a log entry recording that it has committed, then all of its causal predecessors must have committed or must be prepared to commit. The well-formedness and log consistency conditions are enforced by the following judgement forms:

C **logged** S	Well-formed conclave	Fig. 10
$C \vdash C'$	Log consistency	Fig. 11

$$\frac{C \vdash C_1 \quad C \vdash C_2}{C \vdash (C_1 \mid C_2)} \qquad \text{(CONS PAR)}$$

$$\frac{n \notin fn(C) \quad C \vdash C'}{C \vdash (new\ n\ in\ C')} \qquad \text{(CONS NEW)}$$

$$\frac{C \vdash c\{L_1\} \quad C \vdash c\{L_2\}}{C \vdash c\{L_1 \wedge L_2\}} \qquad \text{(CONS CONJ)}$$

$$\frac{C' \in \{c\{P\}, empty, c\{true\}\}}{C \vdash C'} \qquad \text{(CONS NOTLOG)}$$

$$C \vdash c\{PreClosed\} \qquad \text{(CONS PRECLOSED)}$$

$$\frac{\begin{array}{c} C \not\models c' \stackrel{*}{\rightsquigarrow} c \text{ for } c' \in fn(C) - S \\ (C \models c' \stackrel{*}{\rightsquigarrow} c \text{ and } (C \models c'\{PreClosed\} \text{ or } C \models c'\{Closed\})) \text{ for } c' \in S, c' \neq c \end{array}}{C \vdash c\{Closed(S)\}} \qquad \text{(CONS CLOSED)}$$

$$\frac{(C \not\models c'\{c \rightsquigarrow c'\} \text{ or } C \models c\{c \rightsquigarrow c'\}) \text{ for } c' \in fn(C) \quad C \models c\{Closed\}}{C \vdash c\{Undoable\}} \qquad \text{(CONS UNDOABLE)}$$

$$C \vdash c\{PreCommitted\} \qquad \text{(CONS PRECOMMITTED)}$$

$$\frac{\begin{array}{c} C \models c\{Closed(S)\} \\ \forall c' \in S. \exists L' \in \{PreCommitted, Committed, UndoPrepared(c''), UndoAdmin(S')\}. C \models c'\{L'\} \end{array}}{C \vdash c\{Committed\}} \qquad \text{(CONS COMMITTED)}$$

$$\frac{\forall c' \in fn(C). (C \not\models c\{c \rightsquigarrow c'\} \text{ or } C \not\models c'\{Committed\})}{C \vdash c\{Aborted\}} \qquad \text{(CONS ABORTED)}$$

$$\frac{C \models c\{Closed\} \quad C \models c\{Undoable\}}{C \vdash c\{UndoAdministrator(S)\}} \qquad \text{(CONS UNDOADMIN)}$$

$$\frac{C \models c\{Closed\} \quad C \models c\{Undoable\}}{C \vdash c\{UndoPrepared(c')\}} \qquad \text{(CONS UNDOPREP)}$$

Fig. 11. Log Consistency Rules for pik-calculus

In the log consistency rules in Fig. 11, the (CONS CLOSED) rule checks that the set S contains all and only the causal predecessors of c, furthermore that every causal predecessor c' is causally closed or prepared to be causally closed (the latter condition is explained in Sect. 3). The (CONS UNDOABLE) rule requires that, for a conclave to be undoable (as explained in Sect. 5) it must have a log entry for all of its immediate causal successors. So if c' has a log entry recording c as a predecessor, then c must have a corresponding log entry recording c' as its successor. The (CONS COMMITTED) rule requires that the conclave c be causally closed, and that each of its causal predecessors be committed or prepared to commit (as explained in Sect. 4) or be prepared to uncommit (as explained in Sect. 5).

The correctness of the log append rules is verified by the following:

Definition 6.1 A network C_1 is *log-consistent with* C_2 if $(C_1 \mid C_2) \vdash C_1$ is derivable using the derivation rules in Fig. 11. A network C is *log-consistent* if $empty \vdash C$. We sometimes write the latter as $\vdash C$.

Theorem 6.2 *If* $C \vdash C_1$ *and* $C_1 \longrightarrow C_2$, *then* $C \vdash C_2$.

7 Related Work and Conclusions

Numerous process algebras have been proposed as the foundations of programming languages for wide-area applications. Most of the work in the literature is based on mobile computation and mobile code to deal with latency and firewall problems [21,11,12,42,29]. Much of the aforesaid work has focused on access control for mobile computation in networks, as well as tracking the trustworthiness of hosts. Although some work has looked at failures [1,23,41,2], it has assumed a fail-stop model of failures that is not always a good match for programming in asynchronous distributed systems. The synchronous message sending operations of CCS and the pi-calculus require global atomic commitment and therefore are unimplementable in an asynchronous distributed system [20,27]. Palamidessi [39] shows that the leadership election problem can be solved in the pi-calculus, but not in the asynchronous pi-calculus. Herescu and Palamidessi [30] describe a variant of the asynchronous pi-calculus with a probabilistic choice operation, and show that it is possible to implement a leadership election algorithm in this calculus.

Concurrent constraint languages [44,45,8,16] replace message buffers with a global store of constraints, with ask and tell operations for querying the store and adding constraints to the store, respectively. Our model does not replace message buffers in the asynchronous pi-calculus, and indeed we expect that eventually (as alluded to below) remote querying of logs would be implemented using message-passing. Concurrent constraint programs may make the store inconsistent; our operations for modifying stable storage are designed to preserve log consistency, as verified by Theorem 6.2.

Needless to say, transactions and atomic commitment can be implemented in distributed programming languages, and therefore in calculi that are intended to be "kernel languages" for distributed programming. Bruni et al [7] give an implementation of distributed transactions in the join-calculus, using an original atomic commitment protocol. Busi et al [9] propose a formal modelling of transactions in JavaSpaces based on process-calculi techniques. The focus of these efforts is different from the work presented here, which proposes a programming model and a set of abstractions for building different forms of transactions, and different atomic commitment protocols, in global computing environments.

Perhaps the calculus that at least superficially is closest to ours is the join-calculus [21,23]. This calculus allows processes to reflect new process descriptions into the semantics, based on multiset rewriting rules where the multiset contains buffered messages. Related calculi include KLAIM [38], a distributed language based on the Linda primitives [26]. However the intention and therefore the mech-

anisms of the two approaches are quite different. Our use of a multiset of propositions to model stable storage is intended to isolate the communication requirements of fault-tolerance protocols, and the calculus has a predefined collection of rules for adding new log entries, with an emphasis on preserving the consistency of the logs. The distributed join calculus does consider primitives for fault tolerance, but they are based on the fail-stop model that only holds for synchronous distributed systems. An interesting further direction suggested by the join calculus would be to allow applications to define new log entry types, and new rules for adding those log entries to logs during execution. There are interesting security issues with such an idea: What relationships are allowed between new log entry types and existing log entry types, and what log consistency properties could be asserted by applications? What responsibility does an application have to ensure that any log extension rules that it adds preserve log consistency? The join calculus allows new atom types to be defined, by creating new ports, and any process can add atoms (send messages to a port), although receipt of such messages is restricted to the original site. In contrast with the join calculus, new rules for adding log entries of new user-defined types would be global (available to all processes), rather than local as in the join calculus. This is an area for further work.

There are several other directions for further work. One direction is to consider how to extend this model with support for nested transactions [37,35]. The notion of tentative completion and anti-inheritance of locks is particularly interesting in this regard. A notion of equivalence would also be useful for this calculus, particularly a recursive description analogous to the bisimulation method for CCS. Finally, since global computing potentially requires the application to handle remote communication, transaction systems in such an environment cannot assume a secure reliable point-to-point communication system, particularly in running completion protocols. Our approach isolates the remote communication aspects of conclaves to the querying of logs of remote conclaves. We are working on an approach to assigning the application the responsibility of providing the remote communication for this querying, without compromising the security of transitions that affect stable storage. We hope to have the opportunity to report on these developments in subsequent papers.

References

[1] Amadio, R. and S. Prasad, *Localities and failures*, in: P. S. Thiagarajan, editor, *Proceedings of 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 880 in Lecture Notes in Computer Science (1995), pp. 205–216.

[2] Amadio, R. M., *An asynchronous model of locality, failure and process mobility*, in: *COORDINATION'97*, Lecture Notes in Computer Science **1282** (1997).

[3] Amadio, R. M., L. Castellani and D. Sangiorgi, *On bisimulations for the asynchronous pi-calculus*, Theoretical Computer Science **195** (1998), pp. 291–324.

[4] Arnold, K., B. O'Sullivan, R. Scheifler, J. Waldo and A. Wollrath, "The Jini Specification," Addison-Wesley, 1999.

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

[6] Birrell, A., G. Nelson, S. Owicki and E. Wobber, *Network objects*, in: *Symposium on Operating Systems Principles* (1993), pp. 217–230.

[7] Bruni, R., C. Laneve and U. Montanari, *Orchestrating transactions in the Join calculus*, in: *CONCUR 2002, 13th International Conference on Concurrency Theory*, Lecture Notes in Computer Science (2002).

[8] Bueno, F., M. V. Hermenegildo, U. Montanari and F. Rossi, *Partial order and contextual net semantics for atomic and locally atomic cc programs*, Science of Computer Programming **30** (1998), pp. 51–82.

[9] Busi, N., R. Gorrieri and G. Zavattaro, *On the serializability of transactions in JavaSpaces*, in: *ConCoord 2001, International Workshop on Concurrency and Coordination*, ENTCS **54**, 2001.

[10] Cardelli, L., *Abstractions for mobile computation*, in: J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science **1603**, Springer-Verlag, 1999 .

[11] Cardelli, L. and A. Gordon, *Mobile ambients*, in: M. Nivat, editor, *Foundations of Software Science and Computational Structures*, Lecture Notes in Computer Science **1378** (1998), pp. 140–155.

[12] Castagna, G. and J. Vitek, *A calculus of secure mobile computations*, in: *Internet Programming Languages*, Lecture Notes in Computer Science, Springer-Verlag, 1999 .

[13] Chappell, D., *COM+: The next generation*, Byte Magazine (1997).

[14] Cheriton, D. and D. Skeen, *Understanding the limitations of causally and totally ordered communication*, in: *Symposium on Operating Systems Principles*, 1993.

[15] Davidson, S. B., *Optimism and consistency in partitioned database systems*, ACM Transactions on Database Systems **9** (1984), pp. 456–481.

[16] de Boer, F., M. Gabbrielli, E. Marchiori and C. Palamidessi, *Proving concurrent constraint programs correct*, ACM Transactions on Programming Languages and Systems **19** (1998), pp. 685–725.

[17] Detlefs, D., M. Herlihy and J. Wing, *Inheritance of synchronization and recovery properties in avalon/c++*, IEEE Computer (1988), pp. 57–69.

[18] Duggan, D., *Atomic failure in wide-area computation*, in: *Formal Methods in Open Object-Based Distributed Systems (FMOODS)* (2000).

[19] Elmagarmid, A. K., editor, "Database Transaction Models for Advanced Applications," Morgan Kaufmann, 1992.

[20] Fischer, M. J., N. A. Lynch and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM **32** (1985), pp. 374–382.

[21] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the join-calculus*, in: *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* (1996), pp. 372–385.

[22] Fournet, C. and G. Gonthier, *A hierarchy of equivalences for asynchronous calculi (extended abstract)*, in: *ICALP*, Lecture Notes in Computer Science **1443** (1998), pp. 844–855.

[23] Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget and D. Rémy, *A calculus of mobile agents*, in: *7th International Conference on Concurrency Theory (CONCUR'96)* (1996), pp. 406–421, lNCS 1119.

[24] Garcia-Molina, H., D. Gawlick, J. Klein, K. Kleissner and K. Salem, *Modeling long-running activities as nested sagas*, Bulletin of the IEEE Technical Committee on Data Engineering **14** (1991), pp. 14–18.

[25] Garcia-Molina, H. and K. Salem, *Sagas*, in: *ACM SIGMOD International Conference on Management of Data*, 1987, pp. 249–259.

[26] Gelernter, D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[27] Hadzilacos, V., *On the relationship between the atomic commitment and consensus problems*, in: B. Simons and A. Z. Spector, editors, *Fault-Tolerant Distributed Computing*, Lecture Notes in Computer Science **448**, Springer-Verlag, 1990 pp. 201–208.

[28] Haines, N., D. Kindred, J. G. Morrisett and S. M. Nettles, *Composing first-class transactions*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 1719–1736.

[29] Hennessy, M. and J. Riely, *Type-safe execution of mobile agents in anonymous networks*, in: *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science, Springer-Verlag, 1999 .

[30] Herescu, O. M. and C. Palamidessi, *Probabilistic asynchronous π-calculus*, in: J. Tiuryn, editor, *Proceedings of FOSSACS 2000 (Part of ETAPS 2000)*, Lecture Notes in Computer Science (2000), pp. 146–160.

[31] Honda, K. and M. Tokoro, *An object calculus for asynchronous communication*, in: *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science (1991), pp. 133–147.

[32] Korth, H., E. Levy and A. Silberschatz, *Compensating transactions: A new recovery paradigm*, in: *VLDB Conference*, 1990, pp. 95–106.

[33] Lamport, L., *Time, clocks and the ordering of events in a distributed system*, Communications of the ACM **21** (1978), pp. 558–565.

[34] Liskov, B., *Distributed programming in Argus*, Communications of the ACM **31** (1988), pp. 300–312.

[35] Lynch, N., M. Merritt, W. Weihl and A. Fekete, "Atomic Transactions," Morgan-Kaufman, 1994.

[36] Milner, R., *The polyadic π-calculus: A tutorial*, in: F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, Computer and Systems Sciences **94**, Springer-Verlag, 1993 pp. 203–246.

[37] Moss, J. E. B., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT Press, 1985.

[38] Nicola, R. D., G. Ferrari and R. Pugliese, *KLAIM: A kernel language for agents interaction and mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 315–330.

[39] Palamidessi, C., *Comparing the expressive power of the synchronous and the asynchronous pi-calculus*, in: *Proceedings of ACM Symposium on Principles of Programming Languages* (1997).

[40] Pu, C., G. Kaiser and N. Hutchinson, *Split-transactions for open-ended activities*, in: *VLDB Conference*, 1988, pp. 26–37.

[41] Riely, J. and M. Hennessy, *Distributed processes and location failures*, in: *Proceedings of the International Conference on Automata, Languages and Programming*, 1997.

[42] Riely, J. and M. Hennessy, *Trust and partial typing in open systems of mobile agents*, in: *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

[43] Sangiorgi, D., *Asynchronous process calculi: The first-order and higher-order paradigms*, Theoretical Computer Science (1999).

[44] Saraswat, V. and M. Rinard, *Concurrent constraint programming*, in: *Proceedings of ACM Symposium on Principles of Programming Languages*, 1990.

[45] Saraswat, V., M. Rinard and P. Panangaden, *Semantic foundations of concurrent constraint programming*, in: *Proceedings of ACM Symposium on Principles of Programming Languages*, 1991.

[46] Schwarz, R. and F. Mattern, *Detecting causal relationships in distributed computations: In search of the Holy Grail*, Technical Report SFB124-15/92, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (1992).

[47] Siegel, J., D. Frantz, H. Mirsky, R. Hudli, P. deJong, A. Thomas, W. Coles, S. Baker and M. Balick, "CORBA Fundamentals and Programming," John Wiley and Sons, 1996.

[48] Stamos, J. and F. Cristian, *A low-cost atomic commit protocol*, in: *IEEE Symposium on Reliable Distributed Systems*, 1990.

[49] van Renesse, R., *Causal controversy at Le Mont St.-Michel*, Operating Systems Review **27** (1993), pp. 44–53.