Capability-Passing Processes

Tom Chothia

Laboratoire d'Informatique (LIX), École Polytechnique (CNRS), 91128 Palaiseau Cedex, France. tomc@lix.polytechnique.fr

Dominic Duggan

Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030, USA, dduggan@cs.stevens-tech.edu

Abstract

Capability-passing processes model global applications in a way that decouples the global agreement aspects of protocols from the details of how the communications are actually made. It relies on a restricted API or programming language and on the exchange of digital certificates representing *capabilities* to ensure that participants are faithful to a protocol and that outsiders cannot interfere. At the specification level, protocols are reasoned about independently of the underlying communication, using a process calculus with an abstraction of *logs* to isolate the remote state required for such protocols. At the implementation level, protocol steps no longer perform global communication; instead capabilities are used to transmit evidence of remote state, which in turn are used to authorize local log changes (corresponding to protocol steps). In this way, an API for global agreement protocols is defined independently of the underlying communication system.

1 Introduction

Global distributed applications must deal with the fact that the Internet, and other networks, are increasingly becoming a discrete address space, delineated by firewalls, network address translation, independent failures etc. Such applications must also deal with issues of fault tolerance and network security. Fig. 1(a) describes the current state of the art in distributed computing. The typical protocol stack (network layer for routing, transport layer for reliable communication, and application layer) is complicated by the increasing sophistication of the network environment [10]. Thus network and transport protocols are increasingly creaking under the demands of additional tasks such as firewalls, network address translation, and load balancing [6].

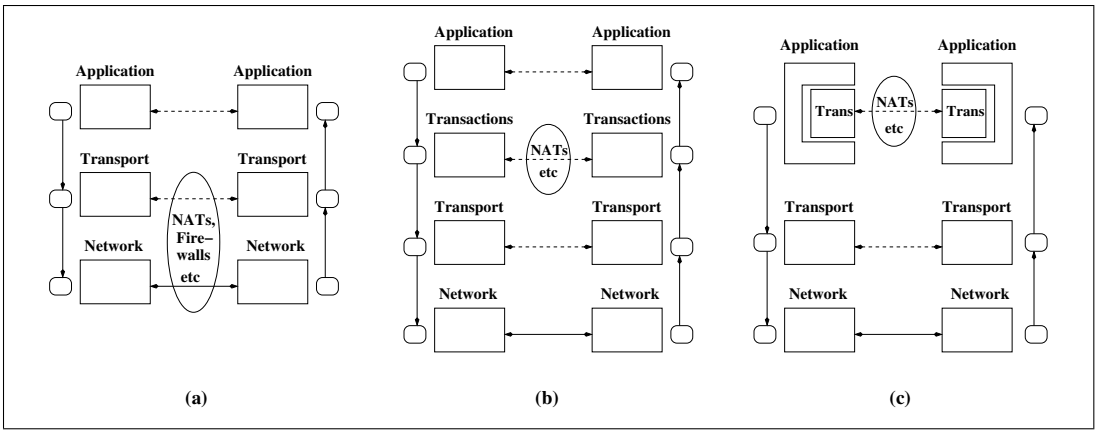

Fig. 1. Structuring protocol layers in global computing

We propose a system in which processes pass proofs of their capabilities between each other, rather than communicating directly. There are two key advantages to this approach. The first is that the user can control how these capabilities are passed between sites, and so navigate any particular network obstacles that the protocol system may not be aware of, such as NAT, VPNs, faulty connections, etc. The second is that by separating the agreement parts of a protocol from the communications aspects we can model both separately.

Our approach is motivated by the fact that any kind of global application will require the design and implementation of protocols for some forms of distributed agreement. Although distributed consensus is in general unsolvable in asynchronous distributed systems [13,16], certain environments may be amenable to assumptions of partial synchrony, and it may be possible for some applications to define a notion of agreement that trades off accuracy for performance. This mediates against building any kind of distributed agreement into the semantics of any language for global applications. Instead we focus on a language for such applications that provides support for designing and implementing protocols for distributed agreement. An implementation of any global agreement protocol will require the ability to deliver messages to different sites. The approach of inserting a transaction layer above the transport layer, as depicted in Fig. 1(b), again raises the problem of complicating a protocol layer with issues that belong to other layers or other parts of the software system.

We advocate the approach depicted in Fig. 1(c), where we move the transaction layer into the application layer. As before, the transaction layer is tasked with ensuring global state consistency conditions (e.g., money not being withdrawn from one account without being deposited into another). The key change is that the transaction layer is decoupled from the navigation of global environments. By separating this layer from the communication functionality, we reduce the size of the trusted computing base required for maintaining global consistency and integrity.

Our approach amounts to a restricted API for changing the logs that represent

the "durable" state of the parties in a distributed system. The restricted API ensures that we only allow changes that preserve the integrity of the overall system state. Complementary to this, the parties must be able to communicate evidence of their state to each other, in such a way that outside parties cannot interfere and cause the system to enter an inconsistent state. While this could be looked at as moving the problem from one place to another, we are in fact, moving the problem of network communication out of the transaction system and into the hands of the programmer. This moves the communication system away from a "one size fits all" solution and allows the user, who will probably known their own network the best, to decide how communication will take place. However, we retain the all the correctness guarantees that would be provided by a transaction system that performed its own communication.

To formulate our method of removing the need for global consensus we return to the lqp-calculus, which we have previously proposed as a language for fault-tolerant global computing [9]. We review this work in Sect. 3. This calculus extends the pi-calculus with *logs*, the action that appends new log entries makes remote checks of other logs at remote sites. In Sect. 5, we show a way of automatically extracting a capability passing version of this calculus, the lqcp-calculus, which only makes local checks. In Sect. 7, we show how we may prove the "correctness" of the capability passing system. Sect. 4 and 6 illustrate our system by modelling the two-phase commit protocol.

2 Related Work

In previous work [8] we proposed an extension of Java with capability passing processes. We discussed a strongly typed calculus as a base for this work but we did not give a type system, semantics or a correctness result.

Berger and Honda [3] have developed a calculus to model two-phase commit. The calculus we present here is a framework that can be used to model a range of agreement protocols, of which two-phase commit is one example, as we have shown in previous work [9]. Numerous process algebras have been proposed as the foundations of programming languages for network applications. Most of this work is based on mobile computation and mobile code to deal with latency and firewall problems [14,5,7,17,23]. Much of this work has focused on access control of mobile computation in networks, as well as tracking the trustworthiness of hosts. Our focus has been on global agreement aspects of distributed computing.

In the distributed systems community there is a lot of work on certificate-based transactions. These systems tend to use certificates to prove identity or authorship, whereas we are proposing the use of certificates to allow the agreement parts of a protocol to be modelled separately from the communication aspects.

Previous work has also investigated techniques for the secure transmission of causal relationships in a distributed system [22,24], and for securing multicast com-

$$
\begin{array}{rcll}
P \in \text{Processes} & ::= & \mathsf{stop} & \mid \quad (P_1 \mid P_2) \\
& \mid & \mathsf{send}\ v_1!v_2 & \mid \quad \mathsf{receive}\ a?x;\ P \\
& \mid & \mathsf{new}\ a;\ P & \mid \quad \mathsf{repeat}\ P \\
& \mid & \mathsf{let}\ \langle \overline{x} \rangle = \langle \overline{v} \rangle\ \mathsf{in}\ P & \\
v \in \text{Value} & ::= & a, b, c, \ldots & \mid \quad u, v, w, x, y, z \ldots \\
& \mid & \langle v_1, \ldots, v_k \rangle &
\end{array}
$$

(a) Syntax of the pi-calculus

$$
\begin{array}{rcll}
P \in \text{Processes} & ::= & \mathsf{logawait}\ c\{\!\{Q(x)\}\!\};\ P & \text{Query log} \\
& \mid & \mathsf{logappend}\ \langle \overline{v} \rangle\ \mathsf{with}\ \textit{rule-name};\ P & \text{Add log entry} \\
N, C, M \in \text{Network} & ::= & c\{P\} & \text{Conclave} \\
& \mid & c\{\!\{L\}\!\} & \text{Log} \\
& \mid & \mathsf{new}\ n;\ C & \text{Scoped name} \\
& \mid & (C_1 \mid C_2) & \text{Composition} \\
L \in \text{Log Entry} & ::= & \mathsf{true} \mid Q(v) \mid (L_1 \wedge L_2) & \\
Q \in \text{Predicate} & ::= & \ldots &
\end{array}
$$

(b) Extensions for the lqp-calculus

Fig. 2. The lqp-calculus

munications [19,15]. The latter work is largely orthogonal to the work considered here. The former work has largely focused on using authentication techniques to prevent the forgery of vector clocks that carry causality information in distributed communications. This is targeted at a lower level than the work presented here. DeLine and Fahndrick have developed the Vault programming language [11] in which describe and statically enforce the correctness of protocols by type checking. The considerations we present here, of a global network programming language that uses proofs of capability to run secure state protocols over a potentially insecure communication system, appears to be novel.

3 The lqp-calculus: Specifying Protocols

The lqp-calculus is based on the asynchronous pi-calculus [20,18], a popular calculus for reasoning about distributed programming languages. The syntax of the pi-calculus is shown in Fig. 2(a). Channel names n are globally unique. In addition to constructs for sending and receiving messages, there are also operations for

generating new channel names, for replicating processes (this can be used to define recursive processes) and for forming the parallel composition of processes.

One of the innovations of the lqp-calculus is to organize processes into process groups; we refer to these process groups as *conclaves*. A conclave has the form $c\{P\}$ where c is the name of a conclave and P is a process. The syntax forces each process in a network to belong to exactly one conclave. There is also a structural equivalence rule for distributing conclaves over parallel composition, $c\{P_1 \mid P_2\} \equiv c\{P_1\} \mid c\{P_2\}$. The processes inside each conclave are active entities, and they can communicate with each other in the standard pi-calculus way. The agreement aspects of a system are modelled by extending the pi-calculus with a notion of logs. These logs are used to explicate the communication requirements of protocols, such as atomic commitment protocols, without committing to how protocol messages should be delivered in global computing environments. Each conclave c has a single log, of the form $c\{\!\{L\}\!\}$, where L is a collection of log entries. A *log entry* has the form $Q(v)$, denoting a log entry asserting the property Q concerning the value v.

The two constructs that allow interaction with logs are logawait and logappend. The logawait construct blocks until a log entry for the conclave name and predicate symbol is in the stable storage represented by the logs. The rule for logawait checks if the required log entry has surfaced:

$$c\{\mathsf{logawait}\ c_0\{\!\{Q(x)\}\!\};\ P\} \mid c_0\{\!\{L \wedge Q(v)\}\!\} \longrightarrow c\{\{v/x\}P\} \mid c_0\{\!\{L \wedge Q(v)\}\!\} \qquad \text{(LogAwait)}$$

The logappend construct is used by a conclave to add new log entries to its own log. It in turn uses one of a collection of named log rewrite rules. These rules define the behaviour of the protocol being modelled, so we will use different sets of rewrite rules for different protocols. A log rewrite rule can check for the presence of log entries in any conclave. However, it can only check for the absence of entries in its own log. An example of such a log rewrite rule might check that a conclave is not already aborted before it commits. Further example log rewrite rules are given in the next section. The following rule is used when a conclave c, which has the log L, uses the logappend action with the log rewrite rule called $rule-name$. The log rewrite rule takes the parameters $\overline{v}$ and states that the predicate $Q(v_0)$ should be added to the log.

$$\frac{(c\{\!\{L\}\!\} \mid N), c \models (\overline{v}) \xrightarrow{\textit{rule-name}} Q(v_0)}{c\{\mathsf{logappend}\ \langle \overline{v} \rangle\ \mathsf{with}\ \textit{rule-name};\ P\} \mid c\{\!\{L\}\!\} \mid N \longrightarrow c\{P\} \mid c\{\!\{L \wedge Q(v_0)\}\!\} \mid N} \qquad \text{(LogAppend)}$$

$$c\{\!\{\varepsilon\}\!\}, c \models (c_0) \xrightarrow{AtSubmit} Submit(c_0) \qquad \text{(AT SUBMIT)}$$

$$c\{\!\{\varepsilon\}\!\} \mid \prod \overline{c_k\{\!\{Submit(c)\}\!\}}, c \models (c_1, \ldots, c_k) \xrightarrow{AtAdmin} Admin(c_1, \ldots, c_k) \qquad \text{(AT ADMIN)}$$

$$c\{\!\{Submit(c_0)\}\!\} \mid c_0\{\!\{Admin(\ldots, c, \ldots)\}\!\}, c \models (c_0) \xrightarrow{AtPrep} Prepared(c_0) \qquad \text{(AT PREP)}$$

$$\frac{L \not\equiv (L' \wedge Committed()),\ (L' \wedge Prepared(\_))}{c\{\!\{L\}\!\}, c \models () \xrightarrow{AtStAbort} Aborted} \qquad \text{(AT STABORT)}$$

$$c\{\!\{Admin(c_1, \ldots, c_k)\}\!\} \mid \prod \overline{c_k\{\!\{L_k \wedge Prepared(c)\}\!\}}, c \models (c_1, \ldots, c_k) \xrightarrow{AtAdmCmt} Committed() \qquad \text{(AT ADMCMT)}$$

$$c\{\!\{L \wedge Prepared(c_0)\}\!\} \mid c_0\{\!\{L_0 \wedge Committed()\}\!\}, c \models (c_0) \xrightarrow{AtPartCmt} Committed() \qquad \text{(AT PARTCMT)}$$

$$c\{\!\{L \wedge Prepared(c_0)\}\!\} \mid c_0\{\!\{L_0 \wedge Aborted()\}\!\}, c \models (c_0) \xrightarrow{AtPartAbt} Aborted() \qquad \text{(AT PARTABORT)}$$

Fig. 3. Log append rules for 2PC in the lqp-calculus

Each log rewrite rule requires some preconditions and adds a log entry to the local log. These rules are specified using judgments of the form:

$$N, c \models (v) \xrightarrow{rule\text{-}name} Q(v_0)$$

where *rule-name* is the name of the rule. The context N must contain the log of the conclave c, and may in addition contain the logs of remote conclaves. The value vector v are parameters for the log rewrite rule (the name of the administrator conclave, for instance). Finally, $Q(v_0)$ is the log entry to be add by the rule.

For some sets of log rewrite rule the logappend and logawait may be the only actions that are needed in order to model a protocol. But in the most general case the use of the send and receive actions together with the log actions allows a network to behave in a much more varied way. Previous work on this calculus, [9] also provides primitives to create new conclaves and logs. Although they add to the expressiveness of the calculus, they do not contribute to the description of capability passing processes so, we omit them here.

$$
\begin{aligned}
\text{Trans}_i \equiv\ & c_i\{\mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtSubmit};\\
& \quad \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPrep};\\
& \quad (\ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartCmt};\ \mathsf{stop}\\
& \quad |\ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartAbt};\ \mathsf{stop}\ )\ \}\\
\text{Admin\_Trans} \equiv\ & c_{adm}\{\mathsf{logappend}\ \langle c_1, c_2\rangle\ \mathsf{with}\ \mathit{AtAdmin};\\
& \quad (\ \mathsf{logappend}\ \langle c_1, c_2\rangle\ \mathsf{with}\ \mathit{AtAdmCmt};\ \mathsf{stop}\\
& \quad |\ \mathsf{logappend}\ \langle c_1, c_2\rangle\ \mathsf{with}\ \mathit{AtStAbort};\ \mathsf{stop}\ )\ \}\\
\\
\text{System} \equiv\ & \text{Admin\_Trans} \mid \mathit{Trans}_1 \mid \text{Trans}_2 \mid c_{adm}\{\!\{\varepsilon\}\!\} \mid c_1\{\!\{\varepsilon\}\!\} \mid c_2\{\!\{\varepsilon\}\!\}
\end{aligned}
$$

Fig. 4. Simple two-phase commit

4 Example: Two-Phase Commit

In this section we give a concrete example of the lqp-calculus by adding log entries and log rewrite rules for the well known and widely deployed two-phase commit protocol [4]. There are five kinds of log entries for two phase commit:

$$Q \in \text{Predicate} ::= \mathit{Submit} \mid \mathit{Prepared} \mid \mathit{Admin} \mid \mathit{Committed} \mid \mathit{Aborted}$$

A $\mathit{Submit}(c)$ entry in a conclave log indicates that the conclave is ready to enter a run of the two phase commit protocol and uniquely identifies c are the administrator conclave. The log entry $\mathit{Admin}(\{c_1,...,c_k\})$ in the log of the administrator conclave records that the conclaves $\{c_1,...,c_k\}$ are the participants in an execution of the two-phase commit protocol. The choice of which conclave will act as the administrator depends on the system being modelled and will perhaps be chosen by a series of pi-calculus communication between conclaves. When one of the participants has successfully completed its task it enters the $\mathit{Prepared}()$ state, after which is cannot abort. If all the participants become $\mathit{Prepared}()$, the administrator may enter the $\mathit{Committed}()$ state. This signals to all the other participants that they may also become $\mathit{Committed}()$. Alternatively, the administrator may become $\mathit{Aborted}()$ at any time before it decides to commit, in which case all the other participants must also abort. The log append rules for two phase commit are given in Fig. 3.

To illustrate the mechanisms introduced above, we show an example using these log append rules. Fig. 4 shows a network where three conclaves, c_{adm}, c_1 and c_2, must either all commit or all abort. A conclave has committed when, and only when, a commit entry has been written to its log. This reflects the fact that sites may fail during runs of protocols, and the state of a fault tolerant system on restarting is given by the contents of the logs.

Once c_1 and c_2 enter the prepared-to-commit state, the system has the form:

$$
\begin{array}{rl}
\text{System} \xrightarrow{*} & c_1\{\ \ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartCmt};\ \mathsf{stop} \\
& \quad\ \ |\ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartAbt};\ \mathsf{stop}\} \\
| & c_1\{\!\{\mathit{Submit}(c_{adm})\ \wedge\ \mathit{Prepared}(c_{adm})\}\!\} \\
| & c_2\{\ \ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartCmt};\ \mathsf{stop} \\
& \quad\ \ |\ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartAbt};\ \mathsf{stop}\} \\
| & c_2\{\!\{\mathit{Submit}(c_{adm})\ \ \wedge\ \mathit{Prepared}(c_{adm})\}\!\} \\
| & c_{adm}\{\ \mathsf{logappend}\ \langle c_1,c_2\rangle\ \mathsf{with}\ \mathit{AtAdmCmt};\ \mathsf{stop} \\
& \quad\ \ |\ \mathsf{logappend}\ \langle c_1,c_2\rangle\ \mathsf{with}\ \mathit{AtStAbort};\ \mathsf{stop} \\
| & c_{adm}\{\!\{\mathit{Admin}((c_1,c_2))\}\!\}
\end{array}
$$

This completes the first phase of the protocol. At this point the conclaves c_1 and c_2 cannot abort or commit until they are notified to do so by the administrator. In the second phase of the protocol, the administrator writes a log entry signaling that it has decided to commit. The participants c_1 and c_2 can then commit, so the system evolves to:

$$
\begin{array}{rl}
\text{System} \xrightarrow{*} & c_1\{\ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartAbt};\ \mathsf{stop}\} \\
| & c_1\{\!\{\mathit{Submit}(c_{adm})\ \wedge\ \mathit{Prepared}(c_{adm})\ \wedge\ \mathit{Committed}()\}\!\} \\
| & c_2\{\ \mathsf{logappend}\ \langle c_{adm}\rangle\ \mathsf{with}\ \mathit{AtPartAbt};\ \mathsf{stop}\} \\
| & c_2\{\!\{\mathit{Submit}(c_{adm})\ \wedge\ \mathit{Prepared}(c_{adm})\ \wedge\ \mathit{Committed}()\}\!\} \\
| & c_{adm}\{\ \mathsf{logappend}\ \langle c_1,c_2\rangle\ \mathsf{with}\ \mathit{AtStAbort};\ \mathsf{stop} \\
| & c_{adm}\{\!\{\mathit{Admin}(c_1,c_2)\ \wedge\ \mathit{Committed}()\}\!\}
\end{array}
$$

It would also have been possible for the administrator conclave to abort, rather than commit. This would then block the rule that c_1 and c_2 are using to try to commit, and so force them to abort.

5 The Capability Passing System: Implementing Protocols

As discussed in the introduction, in the setting of global computing it is unrealistic to maintaining an environment where all conclaves have direct access to each others state. Our answer to this, capability-passing processes, replaces the implicit querying of the state of the logs at remote sites, with "proof" objects that are exchanged as evidence for the capabilities of such a state. Such an object, which is digitally signed by the conclave that generated it, is effectively a proof of a log entry of a particular form. In this section we show how the non-capability passing lqp-calculus can be developed into a capability passing version, the lqcp-calculus.

Any remote checks required to make a log change are replaced by the requirement for a certificate. This means that the protocol actions are now entirely local and all global communication (including the communication of certificates) is modelled by pi-calculus style actions performed by the processes. We extend our domain of values to include these certificates:

$$v \in \text{Value} \quad ::= \quad a,b,c,\ldots \quad | \quad x,y,z,w,\ldots$$
$$| \quad \langle v_1,\ldots,v_k\rangle \quad | \quad S_c[Q(v)]$$

The new kind of value $S_c[Q(v)]$ signifies a "proof" that the conclave c had the entry $Q(v)$ in its log when this certificate was signed. As entries cannot be removed from a log, any conclave that possesses this certificate can conclude that c currently has the entry $Q(v)$ in its log. We do not address the exact method by which the the proof is signed and verified, we consider this an orthogonal issue that is widely addressed elsewhere, .

We alter the logawait, logappend actions and the log append rewrite rules to only query local storage. Any remote queries are replaced by a requirement for a certificate.

From the users point of view logawait performs the single action of querying a log. However, from the point of view of an implementation there are two very distinct versions of logawait. Its first use is to check the current conclaves local log. The second is to check the log of a remote conclave. Our implementation highlights this distinction by having local and global forms of logawait.

A local logawait checks its own log and returns a certificate as a result.

$$c\{\mathsf{logawait}\ Q(x)\ \mathsf{as}\ y;P\} \mid c\{\!\{L \wedge Q(v)\}\!\} \longrightarrow c\{\{v,S_c[Q(v)]/x,y\}P \mid c\{\!\{L \wedge Q(v)\}\!\} \qquad \text{(LogAwait)}$$

In addition to unblocking when the specified log entry is found, the variable y is replaced with a certificate that shows that the specified log entry is present.

There is an implicit assumption that conclaves are always honest when reporting the state of their log. Indeed, this is an assumption made by most transaction systems. The aim of the protocol is usually to guarantee certain results in the presence of an outside attacker or the failure of some given sites. However, if some participants were being dishonest the use of a capability passing system would help to make them accountable by requiring them to sign the state information they distribute.

We extend the syntax of the calculus with a logauth action. This performs a global version of this the logawait action by using a certificate to query the state of a remote log.

$$c\{\mathsf{logauth}\ c_1\{\!\{Q(x)\}\!\}\ \mathsf{with}\ S_{c_1}[Q(v)];P\} \longrightarrow c\{\{v/x\}P\} \qquad \text{(LogAuth)}$$

It should be noted that logauth involves a dynamic verification of the signed proof i.e. the check that the conclave whose log we wish to check is indeed the conclave that signed the certificate and that the certificate does indeed contain the required log entry. If this check fails the construct will block indefinitely. This is similar to encryption and decryption in the spi-calculus [1].

The old logappend action passed the surrounding environment to the log rewrite

rule and so gave it direct access to all logs. This is exactly the kind of global operation that we wish to avoid. The capability passing (LOGAPPEND) rule only accesses the local log. All the global state information is passed to the log rewrite rules in the form of certificates that prove the required remote capabilities ($\overline{v'}$ is the following rule).

$$\frac{c\{\!\{L\}\!\}, \overline{v'}, c \models (\overline{v}) \xrightarrow{\textit{rule-name}} Q(v_0)}{\begin{array}{l} c\{\mathsf{logappend}\ \langle \overline{v} \rangle\ \mathsf{with}\ \textit{rule-name}\ \ \mathsf{and}\ \langle \overline{v'} \rangle; P\} \mid c\{\!\{L\}\!\} \longrightarrow \\ \qquad\qquad c\{P\} \mid c\{\!\{L \wedge Q(v_0)\}\!\} \end{array}}\ (\textsc{LogAppend})$$

The lpq-calculus uses different log rewrite rules for different transactions. To make the transition to the capability passing system complete we give a general method, which can be used to remove the global checks from any log rewrite rule.

This removal of the remote checks is absolutely key to our approach. It removes the requirement for difficult to implement, global querying of state and gives the application layer processes complete control over the methods they use to route communication. Hence removing the need for a bloated trusted computing base to handle secret data.

Recall from Sect. 3, that the preconditions of log rewrite rules can make three kinds of check. They may check for the presence of an entry in the local log, they may check for the absence of a entry in the local log or they may check for the presence of an entry in a remote log. A rewrite rule is adapted to pass capabilities by replacing each remote check by a requirement for a certificate proving that whatever was being checked is true. We use the two phase commit rewrite rules from the last section to illustrate this method.

The (AT PREP) rule is used by a process that is already part of a two phase commit transaction and wishes to enter the "prepared to commit" state. The old version of this rule in Fig. 3 checks the local log to ensure that the conclave is part of a transaction run and then makes a remote check to make sure that the conclave that is playing the part of the administrator has added the Administrator entry into its own log, including the current transaction as part of the run. This remote check is replaced by the requirement for a certificate, to give the following rule:

$$c\{\!\{\textit{Submit}(c_0)\}\!\}, S_{c_0}[\textit{Admin}(\ldots, c, \ldots)] \models (c_0) \xrightarrow{\textit{AtPrep}} \textit{Prepared}(c_0)\ (\textsc{At Prep})$$

The (AT ADMIN) rule is used by the administrator conclave to start its run of the two phase commit protocol. The old log rewrite rule takes a vector of conclave

$$c\{\!\{\varepsilon\}\!\} \models (c_0) \xrightarrow{AtSubmit} Submit(c_0) \qquad \text{(At Submit)}$$

$$c\{\!\{\varepsilon\}\!\}, \prod \overline{S_{c_k}[Submit(c)]} \models (c_1,\ldots,c_k) \xrightarrow{AtAdmin} Admin(c_1,\ldots,c_k) \qquad \text{(At Admin)}$$

$$c\{\!\{Submit(c_0)\}\!\}, S_{c_0}[Admin(\ldots,c,\ldots)] \models (c_0) \xrightarrow{AtPrep} Prepared(c_0) \qquad \text{(At Prep)}$$

$$\frac{L \not\equiv (L' \wedge Committed()),\ (L' \wedge Prepared(\_))}{c\{\!\{L\}\!\} \models () \xrightarrow{AtStAbort} Aborted} \qquad \text{(At StAbort)}$$

$$c\{\!\{Admin(c_1,\ldots,c_k)\}\!\}, \prod \overline{S_{c_k}[L_k \wedge Prepared(c)]} \models (c_1,\ldots,c_k) \xrightarrow{AtAdmCmt} Committed() \qquad \text{(At AdmCmt)}$$

$$c\{\!\{L \wedge Prepared(c_0)\}\!\}, S_{c_0}[L_0 \wedge Committed()] \models (c_0) \xrightarrow{AtPartCmt} Committed() \qquad \text{(At PartCmt)}$$

$$c\{\!\{L \wedge Prepared(c_0)\}\!\}, S_{c_0}[L_0 \wedge Aborted()] \models (c_0) \xrightarrow{AtPartAbt} Aborted() \qquad \text{(At PartAbort)}$$

Fig. 5. Log append rules for 2PC in the lqcp-calculus

names and checks that each of those conclaves has recorded the $Submit(c)$ entry in their log, where c is the name of the administrator. So, to make the capability passing rule, the check that all the participants are submitted is replaced by the requirement for certificates from each of the participants saying they are submitted:

$$c\{\!\{\varepsilon\}\!\}, \prod \overline{S_{c_k}[Submit(c)]} \models (c_1,\ldots,c_k) \xrightarrow{AtAdmin} Admin(c_1,\ldots,c_k) \qquad \text{(At Admin)}$$

An important point to note is that the old rewrite rule implied that all the remote conclaves were checked at the same time whereas, this new rule only requires possession of all of the certificates in order for the administrator to proceed. This greatly adds to the flexibility of the system. For instance, if some of the transaction participants are mobile devices that may enter and leave the administrator's range, the capability passing system will allow the administrator to pick up the certificates whenever the participants are in range and commit as soon as it has them all, rather that waiting, possibly forever, until all the participants are in range at the same time.

The complete capability passing rewrite rules for two phase commit are presented in Fig. 5. The automatic style of rule generation means we can produce capa-

bility passing log rewrite rules for any log rewrite rules in the lqp-calculus. In other work, [9] we present log rewrite rules for closure, causality and anti-commitment. Using the methods outline in this section we automatically get capability passing log rewrite rules for these systems too.

6 Example: Two-Phase Commit in a Capability Passing Style

To illustrate capability passing processes we return to the two phase commit example from Sect. 4. The capability passing version of the two phase commit network is giving in Fig. 6. The core actions of both systems are the same but the capability passing system queries its logs after each logappend action and sends proof of its state to the other parties. These proof objects are then used as evidence in further logappend actions.

Here, for the sake of clarity we are assuming that the two participants have direct communication channels to the administrator. In a more interesting system the certificates would have to be passed through a series of network obstacles, such as NAT boxes and firewalls.

After the c_1 conclave records the *Submit* entry in its log, it uses the local form of logawait to generate a certificate and then sends it to the administrator:

$$\begin{array}{l} c_1\{\ \mathsf{logawait}\ \mathit{Submit}(x)\ \mathsf{as}\ y; \\ \quad \mathsf{send}\ \mathit{outchan}_1!y \mid \\ \quad\ \mathsf{receive}\ \mathit{inchan}_1?z; \\ \quad \ldots\} \mid c_1\{\!\{\mathit{Submit}(c_{adm})\}\!\} \end{array} \longrightarrow \begin{array}{l} c_1\{\ \mathsf{send}\ \mathit{outchan}_1!S_{c_1}[\mathit{Submit}(c_{adm})] \\ \quad \mid \mathsf{receive}\ \mathit{inchan}_1?z; \\ \quad \ldots\} \mid c_1\{\!\{\mathit{Submit}(c_{adm})\}\!\} \end{array}$$

The other participant, c_2, will do the same. This provides the c_{adm} conclave with the evidence it needs to declare itself the administrator of this run of two phase commit, which it does by adding the $\mathit{Admin}(c_1,c_2)$ entry to its own log:

$$\begin{array}{l} c_{adm}\{\ \mathsf{logappend}\ \langle c_1,c_2\rangle\ \mathsf{with}\ \mathit{AtAdmin}\ \mathsf{and}\ \langle S_{c_1}[\mathit{Submit}(c_{adm})], S_{c_2}[\mathit{Submit}(c_{adm})]\rangle; \\ \quad \mathsf{logawait}\ \mathit{Admin}(x_1,x_2)\ \mathsf{as}\ z; \\ \quad \mathsf{send}\ \mathit{inchan}_1!z \mid \mathsf{send}\ \mathit{inchan}_2!z \mid \\ \quad \mathsf{receive}\ \mathit{outchan}_1?u;\ \mathsf{receive}\ \mathit{outchan}_2?v;\ \ldots\} \mid c_{adm}\{\!\{\varepsilon\}\!\} \\ \longrightarrow \\ \quad c_{adm}\{\ \mathsf{logawait}\ \mathit{Admin}(x_1,x_2)\ \mathsf{as}\ z; \\ \quad\quad \mathsf{send}\ \mathit{inchan}_1!z \mid \mathsf{send}\ \mathit{inchan}_2!z \mid \\ \quad\quad \mathsf{receive}\ \mathit{outchan}_1?u;\ \mathsf{receive}\ \mathit{outchan}_2?v;\ \ldots\} \mid c_{adm}\{\!\{\mathit{Admin}(c_1,c_2)\}\!\} \end{array}$$

This reduction is performed using the (RED LOGAPPEND) semantic rule from Sect. 5, which in turn makes use of the (RED AT ADMIN) log rewrite rule from Fig. 5. The log rewrite rule examines the two certificates and checks that they do indeed testify that the remote conclaves have submitted to c_{adm} and that they have the correct signatures. These checks are made dynamically and if they fail the process will block. Here, the certificates are correct, so the checks are successful and the conclave continues to reduce. In this simple example, the pattern of adding a

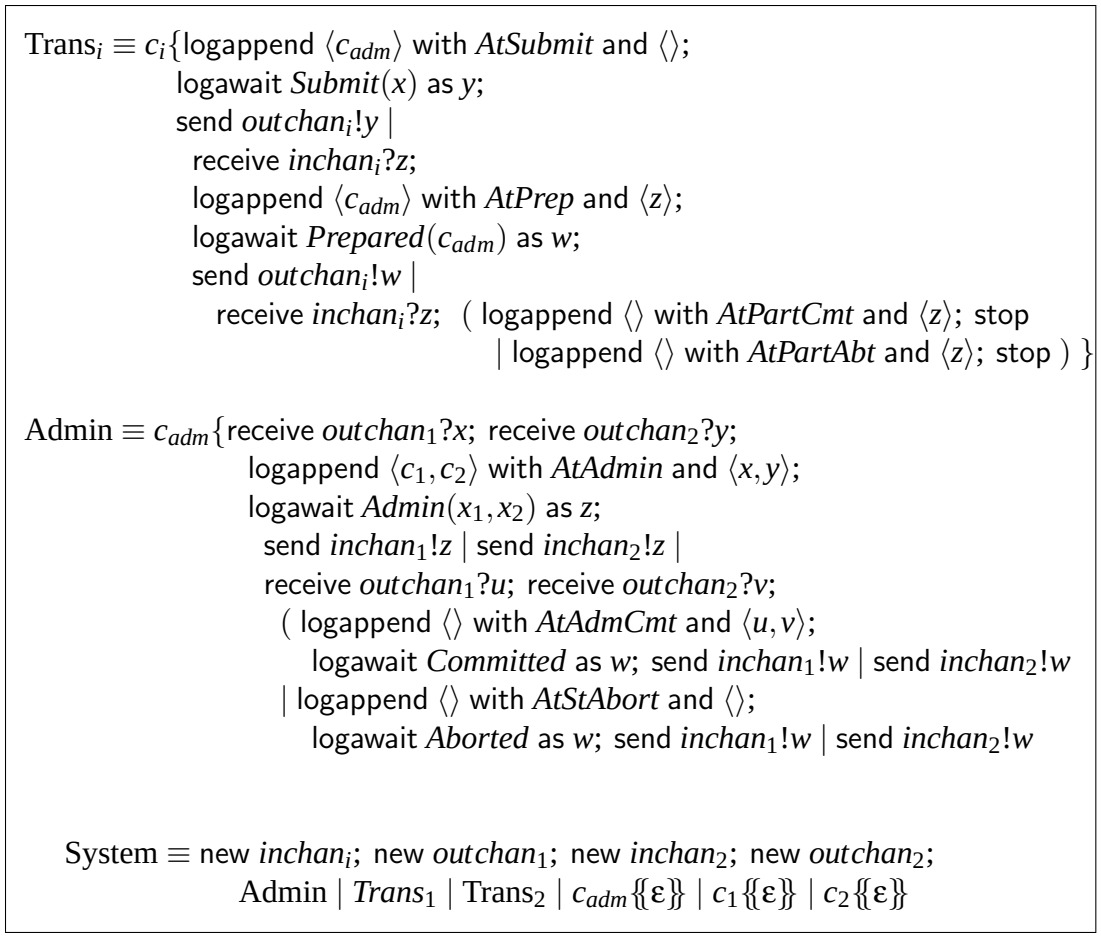

Fig. 6. Simple two-phase commit using proofs

log entry, querying it to get a certificate and then distributing the certificate continues through out the rest of the reduction. In a more realistic system, only certain certificates would be sent to certain conclaves, and each conclave might require different communication protocols to be used. All of which would be modelled by pi-calculus communications between the conclaves.

Once *Trans*$_1$ and *Trans*$_2$ receive the certificate that proves the c_{adm} conclave is in the *Admin* state, they both use it to enter the prepared state. In terms of a run of the two phase commit protocol the receipt of the administrators certificate informs the participants that the transaction has been successfully started. The *Trans*$_1$ and *Trans*$_2$ conclaves enter the prepared state when they have both completed there assigned tasks and are ready to commit. These two conclaves then generate their own certificates and send them to the administrator, which has now reduced to the following:

$$c_{adm}\{\ \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{AtAdmCmt}\ \mathsf{and}\ \langle S_{c_1}[\mathit{Prepared}(c_{adm})], S_{c_2}[\mathit{Prepared}(c_{adm})]\rangle;$$
$$\mathsf{logawait}\ \mathit{Committed}\ \mathsf{as}\ w;\ \mathsf{send}\ \mathit{inchan}_1!w \mid \mathsf{send}\ \mathit{inchan}_2!w$$
$$\mid \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{AtStAbort}\ \mathsf{and}\ \langle\rangle;$$
$$\mathsf{logawait}\ \mathit{Aborted}\ \mathsf{as}\ w;\ \mathsf{send}\ \mathit{inchan}_1!w \mid \mathsf{send}\ \mathit{inchan}_2!w\ \}$$

As with the example process from Sect. 4 the administrator has the choice of whether to commit or abort. As this conclave has *Prepared* certificates from the other two parts of the transaction it can commit. In doing so it blocks the logappend action that is attempting to abort. A certificate testifying to this commitment is then produced and sent to c_1 and c_2, which use it to commit and reach the networks final state:

$$\begin{aligned}
\text{System} \xrightarrow{*}\ & \mathsf{new}\ \mathit{inchan}_i;\ \mathsf{new}\ \mathit{outchan}_1;\ \mathsf{new}\ \mathit{inchan}_2;\ \mathsf{new}\ \mathit{outchan}_2;\\
& c_1\{\ \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{AtPartAbt}\ \mathsf{and}\ \langle S_{c_{adm}[\mathit{Committed}()]}\rangle;\ \ \}\\
& \quad c_1\{\!\{\mathit{Submit}(c_{adm}) \wedge \mathit{Prepared}(c_{adm}) \wedge \mathit{Committed}()\}\!\}\\
& c_2\{\ \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{AtPartAbt}\ \mathsf{and}\ \langle S_{c_{adm}[\mathit{Committed}()]}\rangle;\ \ \}\\
& \quad c_1\{\!\{\mathit{Submit}(c_{adm}) \wedge \mathit{Prepared}(c_{adm}) \wedge \mathit{Committed}()\}\!\}\\
& c_{adm}\{\ \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{AtStAbort}\ \mathsf{and}\ \langle\rangle;\\
& \quad \mathsf{logawait}\ \mathit{Aborted}\ \mathsf{as}\ w;\ \mathsf{send}\ \mathit{inchan}_1!w \mid \mathsf{send}\ \mathit{inchan}_2!w\ \}\\
& \quad \mid c_{adm}\{\!\{\mathit{Admin}(c_1, c_2) \wedge \mathit{Committed}()\}\!\}
\end{aligned}$$

The attempts to abort by c_1 and c_2 block because the certificate that has been passed to these actions does not prove the required state for c_{adm} and so the *AtStAbort* log rewrite rule cannot be applied.

7 Correctness of Capability Passing Systems

Capability passing processes provide an implementation method for systems that wish to query the state of remote parties. It is natural to ask, in what sense is this implementation "correct"? We certainly do not expect a direct relationship between actions of a capability passing process (in the lqcp-calculus) and those of a non-capability passing process (in the lqp-calculus), as capability passing processes will perform a variety of actions that are aimed at passing certificates between conclaves.

Instead, we show that properties, which are base solely on logs and are preserved in the original system, are also preserved in the capability passing system. An example of this kind of judgment is presented in previous work [9] where we define a consistency property on networks that states a range of requirements, such as a log can not be committed and aborted at the same time.

We start this section by defining *supported* networks to be networks with log entries for every certificate. We then show that log alteration in capability passing, supported networks can be mimicked by similar reductions in non-capability passing networks. This fact is used to show our main result .

Definition 7.1 A network is *supported* in the lqcp-calculus, if there exists a log entry for every certificate. i.e $sup(N)$ if and only if for all c, Q, v such that $S_c[Q(v)]$ is a name in N there exists a context C such that $N = C[c\{\!\{\ldots, Q(v), \ldots\}\!\}]$.

As would be expected, this well-formedness property is preserved by reduction:

Lemma 7.2 *In the lqcp-calculus, with any set of log rewrite rules, supported networks reduce to supported networks: if $sup(N)$ and $N \longrightarrow N'$ then $sup(N')$.*

The capability passing semantics can only alter logs in the same way as the non-capability passing semantics. In order to state this formally we define $logs(N)$ to be the erasure mapping of any network to just its logs i.e. $logs(N)$ is the largest sub-term of N that contains only logs. We note that for any lqp-calculus or lqcp-calculus network, the logs of that network are a valid network term for both calculi.

Lemma 7.3 *Given a set of lqp-calculus rewrite rules and their lqcp-calculus versions and a supported capability passing, lqcp-calculus network N such that $N \longrightarrow N'$, using the lqcp-calculus rewrite rules, then there exists a non-capability passing lqp-calculs network M, which does not contain any logs, such that $logs(N) \mid M \longrightarrow logs(N') \mid M'$ using only the non-capability passing semantics and rewrite rules.*

This is analogous to saying that, when using the capability passing lqcp-calculus, an attacker cannot force us to make a change in our logs that is not also possible in the non-capability passing process with its perfect communication. Hence, if it is possible for an attacker to affect a system it has nothing to do with using a capability passing style.

The kind of judgments we are interested in are those based solely on logs. We formally define this class of properties as follows:

Definition 7.4 A judgment on networks $\vdash$ in either the lqp-calculus or the lqcp-calculus is based solely on logs if:

$$(1) \quad \vdash logs(N) \text{ if and only if } \vdash N$$

$$\text{and} \quad (2) \quad \vdash N \text{ and } N \equiv M \text{ implies } \vdash M$$

We can now prove our main result that a judgment based on logs that is preserved by reduction in the non-capability passing calculus is also preserved by reduction on supported networks in the capability passing calculus:

Theorem 7.5 *Any judgment $\vdash$ that is based solely on logs and is preserved by reduction in the non-capability passing lqp-calculus with a given set or log rewrite rules, is also preserved by reduction in the lqcp-calculus, using the capability passing versions of the log rewrite rules.*

8 Conclusions

Our approach to implementing global agreement for distributed protocols is based on a notion of digitally signed "proofs of capability" transmitted between network sites.

This approach has the benefit that a large part of the implementation of the atomic commitment protocol is moved outside of the trusted computing base. In particular none of the primitive operations of the protocol require any remote communication. Communication is left to the application (or some session layer below the application), and no trust is placed on any party outside the protocol implementation.

This paper does not consider the privacy aspects of conclaves giving out signed proofs of their current state. Encrypting the certificates would protect this information. To avoid pushing this into the protocol layer a typed versions of these proofs could be transmitted safely over insecure networks using cryptographic types [12]. Other further work would be to implement a capability passing transaction system as an extension of an already existing package, Maftia [21] and the Java based Jini [2] system are possibilities. It may also be interesting to develop a formal language with which to specify the preconditions of the non-capability passing log rewrite rules and a formal mapping of these rules into their capability passing counterparts.

References

[1] Martin Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

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

[3] Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. In *Proceedings of EXPRESS '00: Expressiveness in Concurrency*, Electronic Notes in Theoretical Computer Science, pages 105–130. Elsevier, 2000.

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

[5] Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[6] B. Carpenter and S. Brim. Middleboxes: Taxonomy and issues. Technical Report RFC 3234, Internet Engineering Task Force (IETF), February 2002.

[7] Guiseppe Castagna and Jan Vitek. A calculus of secure mobile computations. In *Internet Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[8] Tom Chothia and Dominic Duggan. An architecture for secure fault-tolerant global applications. In *Workshop on Principles of Dependable Systems*, 2003.

[9] Tom Chothia and Dominic Duggan. Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3):567–613, 2004.

[10] D. D. Clark and M. J. Blumenthal. Rethinking the design of the Internet: The end to end arguments vs the brave new world. In *Proc. 28th Telecommunications Policy Research Conference*, September 2000.

[11] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[12] Dominic Duggan. Cryptographic types. In *Computer Security Foundations Workshop*, Nova Scotia, Canada, 2002. IEEE Press.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[14] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996. ACM.

[15] L. Gong and N. Shacham. Multicast security and its extension to a mobile environment. *ACM-Baltzer Journal of Wireless Networks*, 1(3):281–295, 1995.

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

[17] Matthew Hennessy and James 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.

[18] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 133–147. Springer-Verlag, 1991.

[19] Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 1997.

[20] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Computer and Systems Sciences*, pages 203–246. Springer-Verlag, 1993.

[21] D. Powell, A. Adelsbach, C. Cachin, S. Creese, M. Dacier, Y. Deswarte, T. McCutcheon, N. Neves, B. Pfitzman, B. Randell, R. Stroud, P. Verissimo, and M. Waidner. MAFTIA (Malicious- and Accidental-Fault Tolerance or Internet Applications). In *Sup. of the Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN2001)*, pages D–32–D–35, 2001.

[22] Michael Reiter and Li Gong. Preventing denial and forgery of causal relationships in distributed systems. In *IEEE Symposium on Research in Security and Privacy*, 1993.

[23] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98: the 25th International Colloquium on Automata, Languages and Programming (Aalborg). LNCS 1443*, pages 695–706. Springer-Verlag, July 1998.

[24] S. W. Smith and J. D. Tygar. Security and privacy for partial order time. In *International Conference on Parallel and Distributed Computing Systems*, 1994.