An Architecture for Secure Fault-Tolerant Global Applications

Tom Chothia
Department of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07040.
tomc@cs.stevens-tech.edu

Dominic Duggan
Department of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07040.
dduggan@cs.stevens-tech.edu

***Abstract*— Applications are increasingly being developed over the Internet, and Internet programming languages seek new abstractions for programming what is in effect a "global computer." The characteristics of the evolving Internet environment have led to new ways of thinking about how to structure such global applications. We advocate a programming language approach to building Internet applications that reflects this change in viewpoint. This approach has a particular emphasis on the demands of fault-tolerant applications, while being centrally concerned with safeguarding security.**

I. Global Computing and Internet Programming

The last decade has seen an explosion in the use of the Internet as a platform for delivering e-commerce (B2B, B2C) to the global economy. The increasing globalization of network computing, particularly for e-commerce, has led to growing interest in programming abstractions for what is in effect a global virtual computer, hence the emergence of *global computing* as a research area [1], as recognized for example by the European Commission FET Proactive Initiative on Global Computing. Today global applications are extending their reach into ubiquitous wireless computing environments.

While RPC and RMI systems have automated the "easy" aspects of network computing, dealing with *independent failures* continues to pose challenges for distributed application developers [2], [3], [4], [5]. Technologies developing for making local networked applications fault-tolerant may not be appropriate to global applications [4]. The characteristics of local and global environments are dissimilar, and numerous impossibility results exist for the latter [6], [7]. Cardelli [1] cites the aspects of global applications that differentiate them from LAN applications. Rather than (as with LANs and the pre-1990s Internet) there being ubiquitious secure point-to-point communication channels between any two sites in the network, the reality in the Internet today is that communication channels must span administrative boundaries imposed by firewalls, proxy servers and network address translators. All of this highlights the demand for programming environments, and particularly programming languages, that support building global applications.

Fig. 1(a) describes the current state of the art in distributed computing over the Internet. The typical protocol stack (of network layer for routing, transport layer for reliable communication, and application layer) is complicated by the increasing sophistication of the network environment [8]. Thus network and transport protocols are increasingly creaking under the demands of additional tasks such as security and firewalls, network address translation, and load balancing [9]. In recognition of the problems that the developing Internet environment is posing, there is a growing suggestion that these additional tasks should be moved out of the lower layer protocols and performed at higher layers in the protocol stack [10], [11], [12], in the application layer or in some layer mediating between the application and the transport, as depicted in Fig. 1(b). In the latter figure we posit a "connection" layer that is part of, or layered underneath, the application layer, as has indeed been suggested in the networks community [21], [22], [12]. For example IETF RFC 3235 offers guidelines for applications that need to navigate across administrative domains delimited by network address translators [11]. We regard this as the essence of global computing and Internet programming languages [1], distinguishing them from local computing and LAN programming languages.

II. Fault Tolerance in Distributed Programming Languages

Achieving atomic failure is recognized as an essential step in building fault-tolerance into distributed applications. Various mechanisms have been proposed for achieving atomic failure. *Transactions* [13] have been adopted as a basic mechanism in some distributed programming languages for supporting the building of fault-tolerant applications. This was pioneered in the Argus distributed language [14], and subsequently in object-oriented distributed languages such as Avalon/C++ and Venari/ML [15], [16]. Transactions are an essential component in various well-known distributed computing platforms, including CORBA OTS, COM MTS, and Java Jini and JavaBeans [17], [18], [19], [20].

Our work is concerned with providing support for fault tolerance for global applications. Specifically we are interested in supporting fault tolerance protocols for such applications, and the implications for these protocols of the restructuring of protocols in Fig. 1. As an example, the Java Jini system [19] has support for fault tolerance, with its provision for transactions. The most important aspect of this support is a set

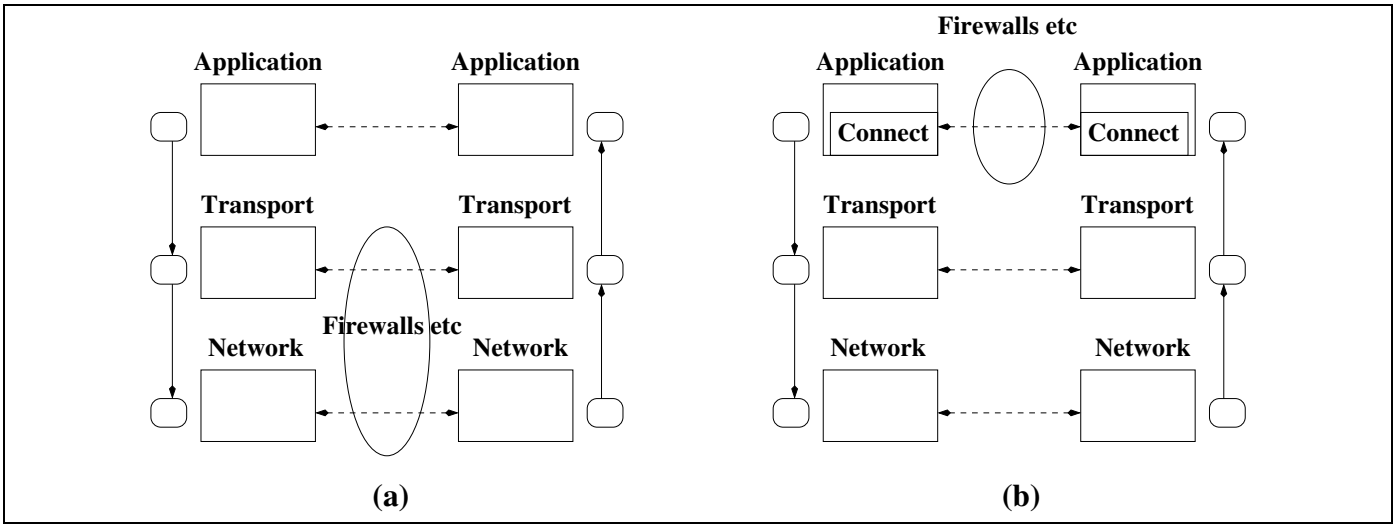


Fig. 1. Structuring protocol layers in global computing

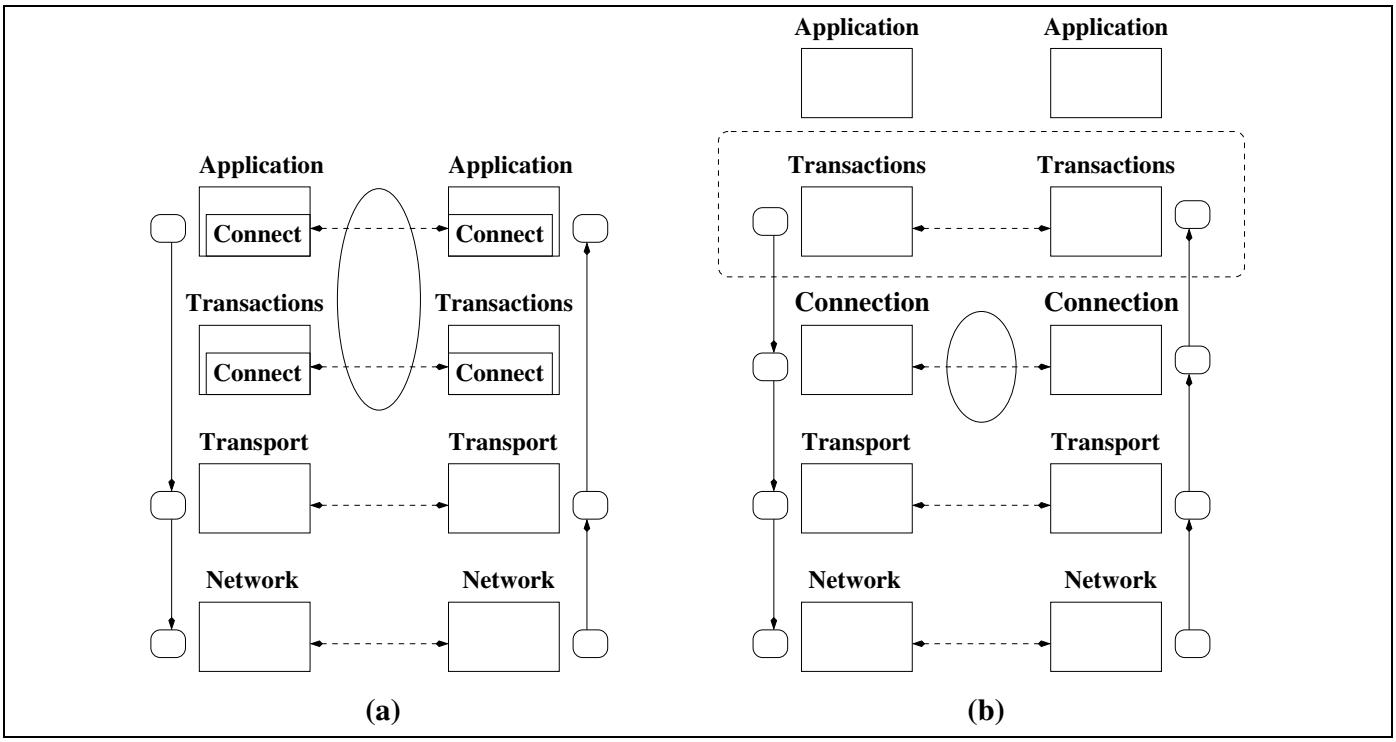


Fig. 2. Structuring fault tolerance protocol layers in global computing

of interfaces for achieving atomic commitment using the two-phase commit protocol, and a default implementation for this protocol. Such transaction systems are typically presented as a middleware layer between the application and the transport layers. However an implementation of the two-phase commit protocol, or for that matter any atomic commitment protocol, requires the ability of the transaction system to deliver protocol messages to different sites in the network. Therefore, as depicted in Fig. 2(a), the approach of layering a transaction system above the transport layer again raises the problem of complicating a protocol layer (transactions) with issues that belong to other layers.

We advocate the approach depicted in Fig. 2(b), where we move the transaction layer above a "connection" layer that handles the delivery of protocol messages across administrative domains [21], [22], [12]. The important point is that the transaction system is decoupled from the navigation of global environments that is the distinguishing characteristic of global computing. There are no "hidden" implicit communications in the model; if nodes in the system collaborating in a fault tolerance protocol need to communicate, they may typically rely on the the application or some "connection" layer to

provide this communication.

There are obvious security issues with this approach, and these security issues are the focus of our work. The essential point is, given that the connection layer may not necessarily be trusted, how do we specify the trust guarantees that must somehow be satisfied for the application as a whole to function correctly? Consider if an e-commerce application is using two-phase commit to effect a transfer of funds from a buyer to a vendor. The trusted part of the system is represented by the databases at the buyer and vendor databases. An inconsistent state is reached if the buyer's database records a withdrawal but there is no record of the deposit in the vendor's database, or vice versa. The untrusted part of the system is the e-commerce application that is contacting the databases as part of its business. Since this application is facilitating the communication between the buyer and vendor databases, navigating between administrative domains, we must consider the communication system as effectively part of the e-commerce application, and therefore similarly untrusted. Our approach is motivated by the security implications of the proposed restructuring of protocols for global computing. It is clear that our approach is not only applicable to fault tolerance protocols, but to the general structuring of global applications where we wish to decouple the communication aspects of the application from the global coordination aspects.

Although we have couched this discussion in terms of a network protocol stack, the point is more general than this, and could be couched more generally in terms of services required and supplied. Our approach decouples the coordination aspects of fault tolerance (e.g., two phase commit protocol) from the communication aspects, and we must somehow specify the trust guarantees required of this communication system. These trust guarantees can then be checked using a combination of static and dynamic checking; the latter will at some point rely on cryptographic operations (digital signing). Our focus in this article is on the specifying the interface between the transaction system and the communication system, and not for example on the connection layer.

III. An Architecture for Global Coordination

An overview of our architecture is provided in Fig. 1. It consists of three layers of calculi, small kernel programming languages that demonstrate the basic features of our approach. These calculi represent different views of a global system:

1) A high-level design view, where the enforcement of global coordination should be isolated from the details of communication between remote parties (the ac-calculus).
2) A high-level "typed" implementation view, that realizes the high-level and relates to it in a precise way that facilitates verification of its correctness (the pac-calculus).
3) A lower-level "untyped" implementation view that represents a generic set of primitives into which the layer above can be compiled (the sac-calculus).

At the core of all of these calculi is a very simple generic message-passing language, the bare minimum that one can expect for any distributed programming language. Our kernel language is the asynchronous pi-calculus [23], [24], [25], a popular calculus for reasoning about distributed programming languages. The calculus' simple structure makes it amenable to reasoning about process equivalence, and this has been used for example in reasoning about security protocols and fault tolerance protocols implemented in the pi-calculus.

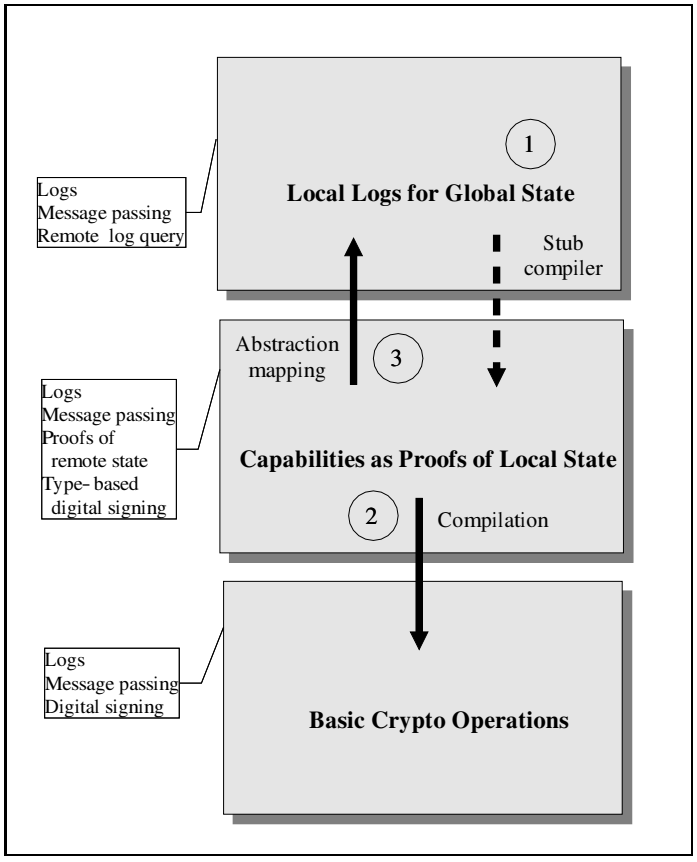


Fig. 3. An Architecture for Secure Fault Tolerant Global Computing

In the following subsections we provide more details about the design-level view and the typed implementation-level view. To make this more concrete, we provide an example API in a hypothetical extension of the Java programming language in Fig. 4.

A. Local Logs for Global State

In terms of the structuring of fault tolerance protocols advocated in Fig. 2(b), the first issue to be addressed in our work is the problem of the implicit communication requirements of fault tolerance protocols, such as two-phase commit. It is desirable to isolate the communication requirements of fault tolerance protocols from the other communication requirements of an application, and to make these requirements explicit as an interface to the "application."

The first of the calculi in our framework is the ac-calculus. This calculus uses a notion of *logs* to achieve the explication of the communication requirements of fault tolerance protocols. Fault tolerant applications can be designed using operations

```
public abstract class Predicate { TransID trans; }
public abstract class LogManager {
   protected void logappend (Predicate p) { ... }
   public Predicate logawait (PredicatePattern p) { ... }
}

public class LogManager2PC extends LogManager {
   final class Committed extends Predicate { }
   final class Aborted extends Predicate { }
   final class Administrator extends Predicate { Set⟨TransID⟩ participants; }
   final class Prepared extends Predicate { TransID administrator; }

   ...
   public void AtAdmCmt () {
      requires: the executing transaction has an Administrator log entry
                all transactions listed as participants have Prepared log entries
      this.logappend (new Committed ());
   }
   public void AtPartCmt () {
      requires: the executing transaction has a Prepared log entry
                the administrator transaction has a Committed log entry
      this.logappend (new Committed ());
   }
   public void AtPartAbt () {
      requires: the executing transaction has a Prepared log entry
                the administrator transaction has an Aborted log entry
      this.logappend (new Aborted ());
   }
}
```

(a) Design Level

```
public class LogManager2PC extends LogManager {
   final class Committed⟨Principal p⟩ extends Predicate⟨p⟩ { }
   final class Aborted⟨Principal p⟩ extends Predicate⟨p⟩ { }
   final class Administrator⟨Principal p⟩ extends Predicate⟨p⟩ { Set⟨TransID⟩ participants; }
   final class Prepared⟨Principal p⟩ extends Predicate⟨p⟩ { TransID administrator; }
   ...
   public void AtAdmCmt⟨Principal p⟩ (Set⟨Prepared⟨p⟩⟩ proofs) {
      requires: the executing transaction has an Administrator log entry
                proofs show: all transactions listed as participants have Prepared log entries
      this.logappend (new Committed⟨thisPrin⟩ ());
   }
   public void AtPartCmt⟨Principal p⟩ (Committed⟨p⟩ proof) {
      requires: the executing transaction has a Prepared log entry
                proof shows: the administrator transaction has a Committed log entry
      this.logappend (new Committed⟨thisPrin⟩ ());
   }
}
```

(b) Implementation Level

Fig. 4. Example of Log-Based API in Java

for querying and modifying log entries. The querying operations may be applied to remote logs, but the modification operations are restricted to local logs. By only allowing local log modifications, we avoid incorporating primitive operations that require distributed agreement, which is provably unimplementable in asynchronous systems [6], [7]. Where it is necessary for a transaction to examine the logs of other transactions, the semantics abstracts away from how this communication is done.

Fig. 4(a) gives an API example using a hypothetical extension of Java with these abstractions. A log is composed of propositions formed from a set of application-specific predicates. In this example we use a set of predicates appropriate to the two-phase set commit protocol. The local state for global coordination is represented by a log, manipulated by a log manager. A log manager has a set of operations for

$A \in$ Type	::=	$Chan[T]$	Channel type
	|	$Prin$	A Principal
	|	$Trans$	A Transaction
	|	$\langle T_1, \ldots, T_n \rangle$	Tuple type
	|	$Set[T]$	Set type

(a) Types in the pi-calculus

$v \in$ Value	::=	a, b, c, n, t	Name
	|	x, y, z, w	Variable
	|	$\langle v_1, \ldots, v_n \rangle$	Tuple
	|	$\{v_1, \ldots, v_n\}$	Set
$P \in$ Processes	::=	$stop$	Stopped process
	|	$send\ v\ on\ v'$	Send message over v'
	|	$receive\ x\ from\ v\ in\ P$	Receive message on v
	|	$let\ \langle x_1, \ldots, x_n \rangle = v\ in\ P$	Decompose tuple
	|	$case\ v\ of\ \{\} \Rightarrow P_1 \mid \{v\} \cup v' \Rightarrow P_2$	
	|	$new\ n : A\ in\ P$	Create new name n
	|	$repeat\ P$	Replication
	|	$(P_1 \mid P_2)$	Parallel composition

(b) Syntax of the pi-calculus with sets

$P \in$ Processes	::=	$logawait\ t\{Q(x)\}\ in\ P$	
	|	$logappend\ \langle \overline{v} \rangle\ with\ rule\text{-}name\ in\ P$	
$C \in$ Transaction	::=	$t\{P\}$	Transaction
	|	$new\ n : A\ in\ C$	Scoped name
	|	$(C_1 \mid C_2)$	Parallel Composition
$N \in$ Network	::=	$a[C]$	Principal
	|	$new\ n : A\ in\ N$	Scoped name
	|	$(N_1 \mid N_2)$	Parallel Composition
	|	$t\{L\}$	Log
$L \in$ Log Entry	::=	$\varepsilon \mid Q(v) \mid (L_1; L_2)$	
$Q \in$ Predicate	::=	$\ldots$	

(c) Extensions for the ac-calculus

Fig. 5. The ac-calculus

appending to the logs. Each such operation represents a step in a protocol for global agreement. Each such operation requires some conditions of the global state, represented by both local log entries and by log entries at remote sites.

The extension of the pi-calculus to the ac-calculus is given in Fig. 5(c). One of the innovations of the ac-calculus is to organize processes into process groups; we refer to these process groups as *transactions.* A transaction has the form $t\{P\}$ where t is the name of a transaction and P is a process. At its simplest, a transaction t groups together a collection of processes. For authentication and trust management purposes, we also group processes by the principal a under which that process executes at run-time, so we have a "two-dimensional" syntax for processes. A process located at its principal has the form $a[C]$ where a is the name of a principal and C its a transaction. It is quite possible for several principals to be executing in the same transaction, e.g., $(a[t\{P_a\}] \mid b[t\{P_b\}])$. Consider for example the Java security model, where the code from various codebases may be executing in a process.

The types for the pi-calculus and the ac-calculus are shown in Figure 5(a). $Prin$ is the type of a principal name and $Trans$ is the type of a name that labels a transaction. $Chan[T]$ is the type of message channels that can carry a value of type T.

The ac-calculus extends the pi-calculus with a notion of logs. These logs are used to explicate the communication requirements of fault tolerance protocols, in particular for atomic commitment protocols, without committing to how protocol messages should be delivered in global computing environments. Each transaction t has a single log, of the form $t\{L\}$, 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. For instance, for the two-phase commit protocol example in Fig. 4, the log predicates have the form:

$$\begin{array}{rcl} Q \in \text{Predicate} & ::= & \mathit{Committed} \mid \mathit{Aborted} \\ & \mid & \mathit{Prepared} \mid \mathit{Administrator} \end{array}$$

Then the log entry $t\{\mathit{Administrator}(\{\overline{t_k}\})\}$ records that $\overline{t_k}$ are the participants in an execution of the two-phase commit protocol administered by t. Eventually a transaction either completes or aborts, and this is recorded by a log entry, either $\mathit{Committed}()$ or $\mathit{Aborted}()$, respectively.

The two constructs that allow interacting with logs are $logawait$ and $logappend$. The *logawait* construct blocks until a log entry for the transaction name and predicate symbol is in stable storage. The *logappend* construct is used to add new log entries. The operations for adding log entries are specified by named rewrite rules. Each rewrite rule requires some preconditions and adds a proposition to the context. These rewrite rules are specified using judgments of the form:

$$(\overline{v}) \models L_s \xrightarrow{\textit{rule-name}} Q(v')$$

where *rule-name* is the name of the rule, L_s is a collection of logs, starting with an entry for the transaction log being appended. The values $\overline{v}$ are parameters in the rule, and $Q(v')$ the proposition added to storage by the rewrite rule. For example, Fig. 6(a) lists the rewrite rules appending log entries for the two-phase commit protocol. Fig. 6(b) gives an example of a single administrator and two participants in the two-phase commit protocol using these log operations.

A transaction t can only modify its own log entries (in the log $t\{L\}$). A transaction can also check its own logs for the *absence* of log entries; for example, a transaction can check that it is not already aborted before committing. A transaction can also check for the presence of log entries for other transactions, though not their absence. The check for preconditions examines the surrounding logs. Where it is necessary for a transaction to examine the logs of other transactions, the semantics abstracts away from how this communication is done.

We have cited two-phase commit as an example of a fault tolerance protocol that our approach should support. But this is only one of many such protocols that applications may attempt to use. Indeed it is clear that our approach is more generally applicable to applications where nodes in a global network need to change state in a coordinated way, with the logic of this coordinated state change decoupled from the details of message-passing and navigation of administrative domains.

Concurrent constraint languages [26], [27], [28], [29] 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 implement remote querying of logs using message-passing, in the lower level pac-calculus. Concurrent constraint programs may make the store inconsistent; our operations for modifying stable storage are designed to preserve log consistency, and this is a proof obligation for any log append rules added by an application.

B. Capabilities as Proofs of Local State

We have at this point isolated the remote communication requirements of our kernel language to being able to query the logs at remote sites. All other communication is left to the application, so for example the application takes responsibility for setting up communication channels and securing communication. This moves much of the details of remote communication out of the language semantics and into software libraries. Besides having the benefit of reducing the size of the trusted computing base (TCB), this also allows applications potentially to choose from different communication libraries based on the network characteristics.

We are still left with the task of showing how to provide remote log querying. Correctness of any of the protocols that we are interested in is based on maintaining a globally consistent state in a distributed system. The state of the system is reflected by the state of the logs, that record permanent state changes (as opposed to ephemeral state changes that may be undone by failures). The challenge is to maintain the logs in a consistent state, while potentially malicious processes are trying to subvert this. As noted earlier, in an e-commerce application, the trusted part of the system is the state of the

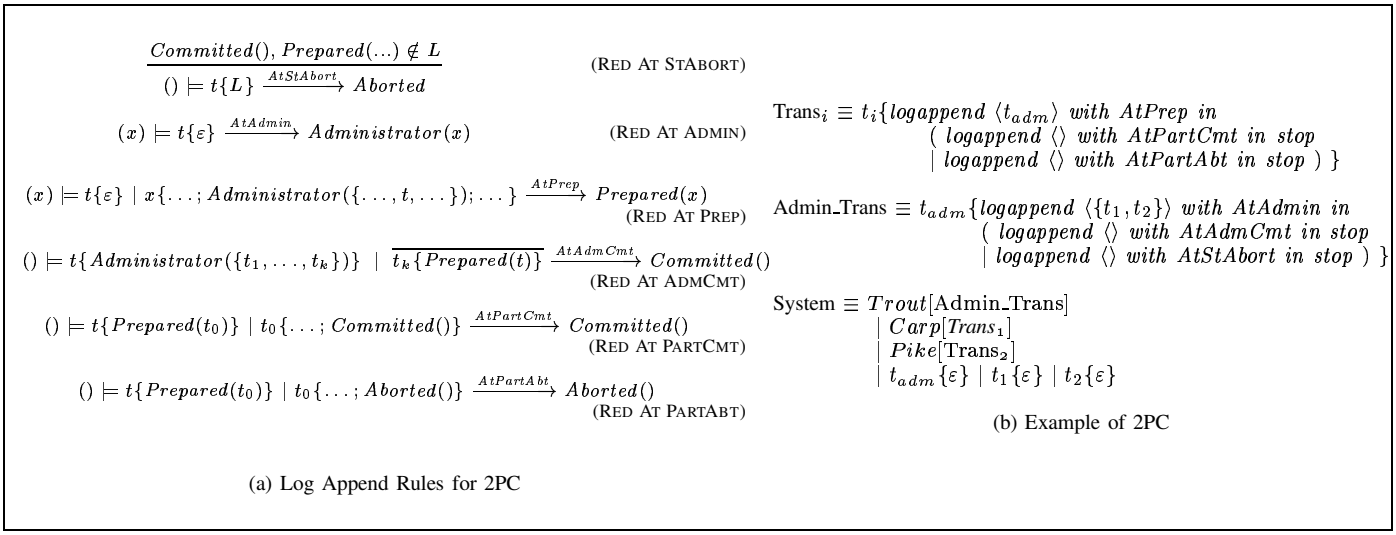


Fig. 6. Example of 2PC for the ac-calculus

logs at the buyer and the vendor. The untrusted part of the system is the e-commerce application that is seeking to access these logs as part of its business, and is providing part of the communication system between the banks.

Our approach is to map the remote log query operations of the ac-calculus down into the existing communication system, removing the implicit assumption in the ac-calculus of point-to-point communication links between any two sites in the network. The challenge is to do this in a way that protects the integrity of the logs from attacks based on "spoofing" protocol messages. We provide an API for querying and modifying logs based on *capabilities as proofs*. The "proofs" in this case are evidence of the state of the logs at remote sites. Such proofs are signed by the principals that generate them, making our approach amenable to the application of trust management techniques to provide control over the ability to change logs.

The ac-calculus is a design calculus for global applications, concentrating on the coordination aspects of such applications. We propose the pac-calculus as an implementation calculus for this design calculus. This latter calculus replaces the implicit querying of the state of the logs at remote sites, with capabilities that are exchanged as evidence for such state. Such a capability, that could be digitally signed by the principal that generated it, is effectively an explicit proof object for a log entry of a particular form. This is made explicit by the type of such a capability, reflecting the log entry that it asserts to be present at a site. The syntax of the pac-calculus is given in Fig. 7.

A proof is a name (just as with transaction names and channel names). The type of this name is a log entry, signed by the principal providing the proof (not the principal that added the log entry). Therefore the type of a proof n has the form $a[t\{Q(v)\}]$, signifying that this is a proof, signed by the principal a, that the transaction t has a log entry $Q(v)$; and we have the type membership:

$$n \in a[t\{Q(v)\}].$$

The *logawait* construct now only queries local storage. In addition to unblocking when the specified log entry is present, it also creates and returns the proof that the specified log entry is present. Proof generation is delayed until the execution of *logawait* because the proof is signed by the principal generating the proof, rather than the principal that added the log entry. Wherever a *logappend* rule requires knowledge of the state of remote logs, the rewrite rule is modified to require explicit proofs of the state of these remote logs, as part of its input. The $logappend$ construct has a different type rule for each of the different rewrite rules. These type rules ensure that appropriate proofs have been provided. Log alterations that require the presence of particular names in remote logs, as well as particular states, ensure their presence by way of additional variables in $logcommit$'s arguments.

For several reasons we need to elide some of the details in the types of a capability. Most importantly for distributed computing, we do not expect that in an untrusted network environment we can obtain secure channels with such precise types. Our approach instead is to introduce *packets* to elide information in the type of a proof. We inject a proof into such a packet; some of this injection may involve actual digital signing. We then transmit this packet essentially untyped using the communication system. At the receiving site, we extract the proof from the packet; an implementation of this extraction will involve actual digital signature authentication. In effect, we are using proofs and packets to construct secure channels over insecure communication channels.

In summary, in the pac-calculus we have several gradations of "capability:"

1) An atomic proof of type $a[t\{Q(v)\}]$.
2) A *witness packet* of type $WitPkt[a, Q]$ that elides some details of the proof type.
3) A signature packet of type $SgnPkt[Q]$ that represents actual digital signing, by eliding the identity of the signing principal a. A receiver must then authenticate

$B \in$ Type	::=	$a[t\{Q(v)\}]$	Proof Type
$v \in$ Values	::=	$witPkt_Q\ p_2\ v\ p_3$	Witness packet
	|	$sgnPkt_Q\ p_1\ v$	Signature packet
$P \in$ Processes	::=	$logawait\ x\ for\ t\{Q(x)\}\ in\ P$	
	|	$logappend\ \langle\overline{v}\rangle\ with\ rule\text{-}name\ in\ P$	
	|	$sign\ v\ is\ x : SgnPkt[Q]\ in\ P$	Hide principal
	|	$auth\ v\ with\ v'\ is\ x\ in\ P$	Expose principal
	|	$open\ v\ is\ x, y, z\ in\ P$	Expose witness
$C \in$ Transaction	::=	$t\{P\}$	Transaction
	|	$new\ n : B\ in\ C$	Scoped name
	|	$(C_1 \mid C_2)$	Parallel Composition
$N \in$ Network	::=	$a[C]$	Principal
	|	$new\ n : B\ in\ N$	Scoped name
	|	$(N_1 \mid N_2)$	Parallel Composition
	|	$t\{L\}$	Log
$L \in$ Log Entry	::=	$\varepsilon \mid Q(v)\ \mathbf{prf}\ \overline{n} \mid (L_1\ ;\ L_2)$	
$Q \in$ Predicate	::=	$\ldots$	

Fig. 7. Extensions to the pi-calculus for the pac-calculus

the signature to re-expose the principal a in the type.

A witness packet can be built directly by a process. A signature packet on other hand requires access to the private signing key of a principal. Such keys are not manipulated explicitly by processes. If a principal name is transmitted in a message, in the compilation of a pac program, this principal name is translated as the public key of the principal. The private key for a principal a is only available within a process of the form $a[t\{P\}]$, and is used implicitly in the *sign* construct for signing a witness packet (resulting in a signature packet). We also have constructs for exposing the identity of the principal in a signature packet (the *auth* construct), and exposing the witness value and transaction name in a witness packet (the *open* construct). For the *auth* construct for authenticating the signature in a packet, there is an explicit check that the public key b being used to authenticate the packet matches the private key b used to digitally sign the packet. This equality check in the pac-calculus can be translated to actual digital signature authentication in an underlying implementation calculus.

Fig. 4(b) continues the example from Fig. 4(a), replacing the remote log querying in the log-changing operations with extra arguments that represent "evidence" for log entries at remote sites. Predicate types, the types of such proofs, are parameterized by the names of signing principals. Suppose `Admin` is the name for a trusted principal for the administrator in the two-phase commit protocol. Then a participant will accept a proof of type `Committed⟨Admin⟩`, i.e., a proof that the administrator has decided to commit the transaction:

```
class Chan⟨A⟩ { A receive (); }
```

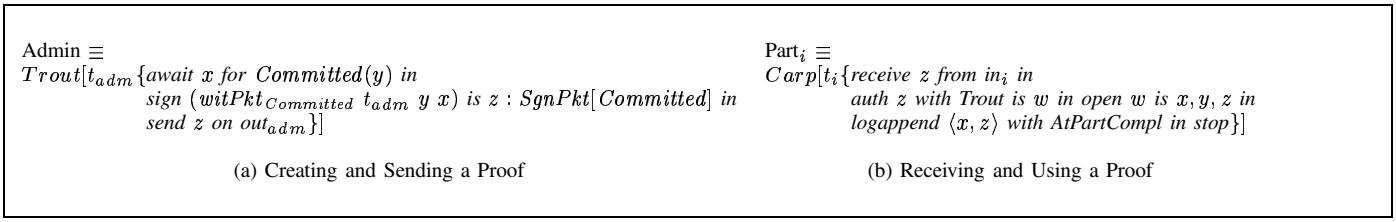
```
// Participant
LogManager2PC lm;
Chan⟨Committed⟨Admin⟩⟩ chan;
Committed⟨Admin⟩ pf = chan.receive();
if (pf.trans==adminTrans) lm.AtPartCmt(pf);
```

If a 'secure channel" (i.e., one with the explicit signed type above) cannot be obtained, then we use type-based cryptographic operations to build a secure channel. Such operations use upcasting and downcasting at the source language level to represent digital signing and authentication, respectively, at the implementation level [36]:

```
// Participant
LogManager2PC lm;
Chan⟨Committed⟩ chan;
Committed msg = chan.receive();
Committed⟨Admin⟩ pf = (Committed⟨Admin⟩)msg;
if (pf.trans==adminTrans) lm.AtPartCmt(pf);
```

We have simplified the description of the type system here. In particular we need to stratify the type system so that proof generation is a "trusted" network operation. Processes only have access to this proof-generation operation through the *logawait* operation, so in a well-typed program proofs cannot be forged. Processes accepting packets from untrusted processes must use trust management techniques, based on the signature of the proof, to determine if the underlying proof should be accepted.

We now show how the example in Fig. 6(b) can be modified to use explicit proofs to replace the implicit querying of remote logs. For reasons of space, we restrict the example



(a) Creating and Sending a Proof

(b) Receiving and Using a Proof

Fig. 8. 2PC with Proofs

to the case where one of the participants commits after receiving notification from the administrator. The example is given in Fig. 8. The administrator side of this protocol requires querying the local logs to generate a proof that the administrator has decided to commit the transaction as a whole. Once the proof is obtained, it is bundled in a witness packet and then a signature packet and the result output on a channel out_{adm}. The communication system, which may involve upper layer protocols for navigating firewalls and network address translators, for example, eventually delivers the message output on out_{adm} to the input channel in_i for the ith participant. The latter checks the digital signature to make sure the packet was indeed sent by *Trout*, extracts the transaction identifier t_{adm} and proof n from the witness packet, then uses the latter to justify appending a log entry recording the decision of this participant to commit.

Since the ac-calculus provides the abstraction of logs, a default translation from the ac-calculus to the pac-calculus can use a middleware of message communication channels, name servers, and capabilities as proofs, to implement the remote querying available in the ac-calculus. If we wish to reason in the ac-calculus about security attacks on applications built using this infrastructure, we can explore the techniques developed by Abadi et al [30], [31] for having the compiler insert firewall code into ac-calculus programs in the process of translating them to the pac-calculus. We refer to this as the "trusted compiler" approach to securing network communication; it is basically moving some of the firewall code out of the OS protocol stack and into the compiler. We identify three problems with pursuing the trusted compiler approach:

1) First, because of the full abstraction requirement for compilation, there is still a fair gap between the theory of the trusted compiler approach and a practical realization. For example the compiler must be able to insert code that prevents denial-of-service attacks, a very strong requirement when there are no restrictions on the (physical or virtual) network topology or the intermediate hosts that may be enabling communication.
2) Second, the trusted compiler approach, as its name suggests, makes the compiler that produces the firewall code part of the TCB. This enlarges the TCB with a fairly sophisticated compiler. In contrast, compiling languages such as the ac-calculus and the pac-calculus, without inserting firewall code, is fairly straightforward and the compilers correspondingly simpler.
3) Third, the trusted compiler approach assumes a point-to-point communication infrastructure that can establish communication between any points in the network. In contrast, global (Internet) computing suggests a programming environment where, because of administrative boundaries and physical characteristics, this assumption may be strained or broken.

Some default translation from the ac-calculus to the pac-calculus would undoubtedly be useful in a programming environment. However we advocate a different approach in general, based on our consideration of the characteristics of Internet computing environments. We advocate treating a program in the ac-calculus as a *specification* of a distributed application. The full implementation of such a program must provide a mechanism for communicating "evidence" across the network. Such an implementation can be provided using a combination of off-the-shelf libraries. The important point is that the communication system is taken out of the TCB. Using explicit proofs, the TCB ensures that only valid state changes happen, but it is left to the application to ensure that these proofs are delivered between sites. The correctness of implementations in the pac-calculus, for designs in the ac-calculus, is provided by an abstraction mapping from programs in the former to programs in the former.

Our approach to supporting fault tolerance protocols in global computing is based on a notion of digitally signed "proofs," transmitted between network sites, of local state changes. This is related to the notion of capabilities that has been implemented in some distributed systems [32], [33], [34], [35]. In our context, we use the type system to express the global properties that capabilities assert, and use typing to reason about the correctness of programs that use these capabilities. Such typed capabilities can be transmitted safely over insecure networks by adapting the compilation scheme for cryptographic types [36]. This approach of typed capabilities and digital signatures has the benefit that none of the primitive operations of the protocol require any remote communication; they are all strictly local operations, that examine and modify the local logs. 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.

IV. Conclusions

We have provided motivation and a broad overview for an architecture that we are developing to support fault tolerant applications running in some form of secure fashion over the Internet. In other work we have reported on a specific instanti-

ation of the ac-calculus [37], indicated by (1) in Fig. 3. This shows how various mechanisms for tracking dependencies, implementing atomic commitment and anticommitment (for optimistic computation) can be formulated as log types and rules for appending to logs. We have also described a type-based approach to cryptographic operations [36], indicated by (2) in Fig. 3, where cryptographic guarantees are expressed in the type system. The types can be used to express the security guarantees that the environment can provide, e.g., the security guarantees that a lower-layer service can provide to upper layers. This allows these guarantees to be checked statically, with resort to expensive dynamic checking only necessary when the environment is insufficiently secure. The cryptographic operations of encryption/signing and decryption/authentication are then represented in this typed API as upcasting (widening) and checked downcasting (narrowing) of types. We hope to have the opportunity to report on the pac-calculus and its relation to the ac-calculus, indicated by (3) in Fig. 3, in a subsequent paper. An implementation is also planned.

ACKNOWLEDGEMENTS

Thanks for Dan Duchamp and Rebecca Wright for helpful conversations. Thanks to the anonymous reviewers for their excellent feedback.

REFERENCES

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

[2] B. Lampson, "Computer system research: Past and future," in *Symposium on Operating Systems Principles*, Charleston, South Carolina, 1999, invited talk.

[3] A. Black, "Distributed objects: The next ten years," in *Foundations of Object Oriented Languages*, Portland, Oregon, 2002, invited talk.

[4] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Mobile Object Systems*, ser. Lecture Notes in Computer Science, J. Vitek and C. Tschudin, Eds. Springer-Verlag, 1997, pp. 49–64.

[5] B. Merrill, "Distributed objects: .NET versus Java," panel Discussion at OOPSLA 2002.

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

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

[8] 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.

[9] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," Internet Engineering Task Force (IETF), Tech. Rep. RFC 3234, February 2002.

[10] K. Moore, "On the use of HTTP as a substrate," Internet Engineering Task Force (IETF), Tech. Rep. RFC 3205, February 2002.

[11] D. Senie, "Network address translator (nat)-friendly application design guidelines," Internet Engineering Task Force (IETF), Tech. Rep. RFC 3235, January 2002.

[12] D. J. Duchamp, "The Discrete Internet and what to do about it," in *New York Metropolitan Area Networking Workshop*, September 2002.

[13] B. Lampson, "Atomic transactions," in *Distributed Systems–Architecture and Implementation*, ser. Lecture Notes in Computer Science, B. Lampson, M. Paul, and H. Siegert, Eds. Springer-Verlag, 1981, vol. 205, pp. 246–285.

[14] B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp. 300–312, March 1988.

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

[16] N. Haines, D. Kindred, J. G. Morrisett, and S. M. Nettles, "Composing first-class transactions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1719–1736, November 1994.

[17] J. Siegel, 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.

[18] D. Chappell, "COM+: The next generation," *Byte Magazine*, December 1997.

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

[20] G. Hamilton, "JavaBeans API Specification v1.01," Sun Microsystems, Tech. Rep., 1997.

[21] D. Kristol and L. Montulli, "HTTP state management mechanism," Internet Engineering Task Force (IETF), Tech. Rep. RFC 2965, November 2000.

[22] S. Spero, "Session control protocol (scp)," World Wide Web Consortium (W3C), Tech. Rep., 2002, http://www.w3.org/Protocols/HTTP-NG/http-ng-scp.html.

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

[24] K. Honda and M. Tokoro, "An object calculus for asynchronous communication," in *European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1991, pp. 133–147.

[25] R. M. Amadio, L. Castellani, and D. Sangiorgi, "On bisimulations for the asynchronous pi-calculus," *Theoretical Computer Science*, vol. 195, no. 2, pp. 291–324, 1998.

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

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

[28] F. Bueno, 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*, vol. 30, pp. 51–82, 1998.

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

[30] M. Abadi, C. Fournet, and G. Gonthier, "Secure communications processing for distributed languages," in *IEEE Symposium on Security and Privacy*, 1999.

[31] ——, "Authentication primitives and their compilation," in *Proceedings of ACM Symposium on Principles of Programming Languages*, 2000.

[32] J. B. Dennis and E. C. van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, pp. 143–155, March 1966.

[33] R. S. Fabry, "Capability-based addressing," *Communications of the ACM*, vol. 17, pp. 403–412, July 1974.

[34] W. A. Wulf, E. S. Cohen, W. M. Corwin, A. K. Jones, R. Levin, C. Pierson, and F. J. Pollack, "HYDRA: The kernel of a multiprocessor operating system," *Communications of the ACM*, vol. 17, pp. 337–345, June 1974.

[35] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum, "Experiences with the Amoeba distributed operating system," *Communications of the ACM*, vol. 33, pp. 46–63, December 1990.

[36] D. Duggan, "Cryptographic types," in *Computer Security Foundations Workshop*. Nova Scotia, Canada: IEEE Press, 2002.

[37] ——, "Abstractions for fault-tolerant global computing," in *Foundations of Wide-Area Network Computing (FWAN)*, ser. Electronic Notes in Theoretical Computer Science (ENTCS), vol. 66, no. 3, 2002, extended abstract.