# Abstractions for Fault-Tolerant Wide-Area Network Programming Languages

Dominic Duggan

Department of Computer Science,
Stevens Institute of Technology,
Castle Point on the Hudson,
Hoboken, New Jersey 07030.
dduggan@cs.stevens-tech.edu

**Abstract.** Wide-area computation is a recent area of research, concerned with developing abstractions for programming the Internet. The organizing principle for this work has been mobile computation, allowing code or processes to be transported across the network. This paper reviews some of the key developments in wide-area computation, and considers the ATF Calculus, a kernel language for wide-area network programming with atomic failure as its organizing principle.

## 1 Introduction

Networks and distributed applications have become an indispensable and ubiquitous part of the modern computing infrastructure. The move from centralized mainframes to networks of workstations connected to each other and server machines, has been driven by the economics of the latter paradigm. The World Wide Web and electronic commerce (e-commerce) have driven the movement to connect these local-area networks into the Internet.

Concommitant with the connecting of commercial LANs to the Internet has been the deployment of *firewalls* to protect these LANs from hostile intruders [15]. A firewall effectively partitions a network based on levels of trust, with more trusted hosts inside the firewall. The trusted hosts inside the firewall constitute an *administrative domain* defined by the firewall. A commercial intranet may be protected by an external firewall, while at the same time being partitioned by internal firewalls (for example, protecting competitive corporate divisions, or simply enforcing the principle of "least access necessary"). This gives rise to hierarchial nesting of administrative domains, as described by Fig. 2.

The Internet is effectively a collection of federated wide-area networks. Although firewalls have defined a de facto partitioning of the Internet into administrative domains, as described above, firewalls themselves are becoming obsolete, precisely because of the need of hosts to interoperate with other hosts outside their administrative domains. For example, it is by now becoming commonplace to expect to be able to access electronic mail while outside the company's physical firewall, say using a laptop computer in a hotel room. This access is facilitated by cryptographic protocols that negotiate access through the firewall, and allow information to be transferred in a secure manner using encryption. The transfer of encrypted data has the side effect of undermining the usefulness of firewalls for mediating transfers across the administrative domain. An alternative point of view is to regard these cryptographic protocols as defining administrative domains, using authentication and encryption to build *virtual private networks* (VPNs). From this point of view, the person accessing e-mail from a laptop computer in a hotel room is behind the "virtual" firewall, by dint of cryptographic keys that may be "baked" into the laptop.

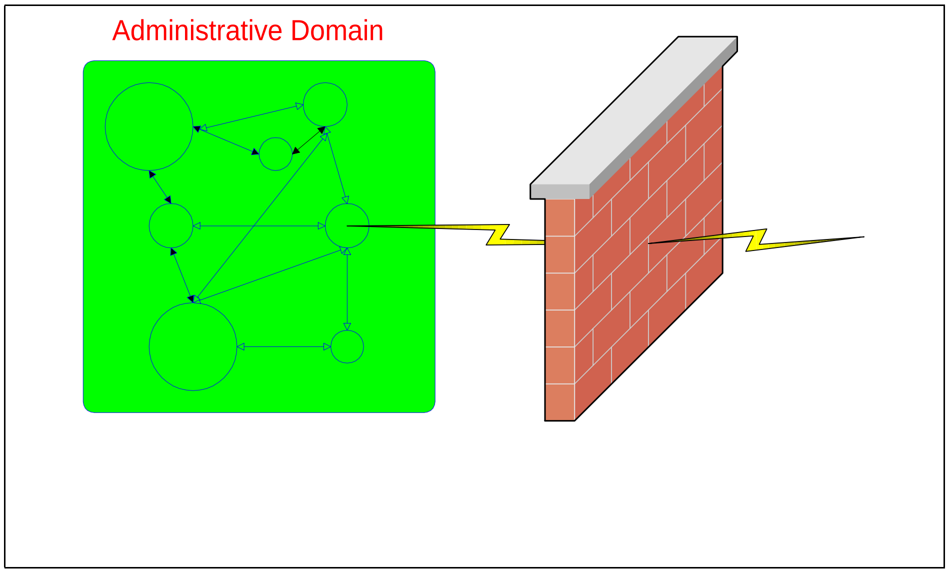

**Fig. 1.** Administrative Domain Defined by a Firewall

Wide-area networks pose other languages for network programming environments. Failures, although possible in LANs, are much more likely in WANs. Communication failures (network partitions) in particular are comparatively rare in LANs. Many protocols for failure handling in LANs assume the *failstop* model for host failures, where a host announces its failure before crashing. This means that protocols can be devised that rely on being able to determine if a host has crashed and respond accordingly. This is possible because a LAN is effectively a *synchronous system*: both message delays and processing times can be bounded from above, so the failure to respond to a message request within an adequate period of time can be interpreted as a failure.

In contrast WANs and the Internet constitute *asynchronous systems*, where no upper bound can be placed on message transmission delays or processing times. For example, although it is possible to devise algorithms for achieving atomic consensus in synchronous systems, this is known to be impossible in asynchronous systems [19]. This impossibility result extends to achieving non-blocking protocols for atomic commitment in asynchronous systems [47, 24, 22]. This difference in the semantics of failures in LANs and WANs is reflected in the Jini infrastructure for distributed programming promulgated by Sun Microsystems [5]. Although LAN programming environments have striven to make the network "transparent," the Jini architecture deliberately exposes the network in the way it handles failures: a client with access to a remote resource may lose that access because of communication failures, since such failures can block "keep-alive" messages sent to the resource. This reflects a deliberate philosophy on the part of the Jini designers that WAN environments should not be hidden from applications [49].

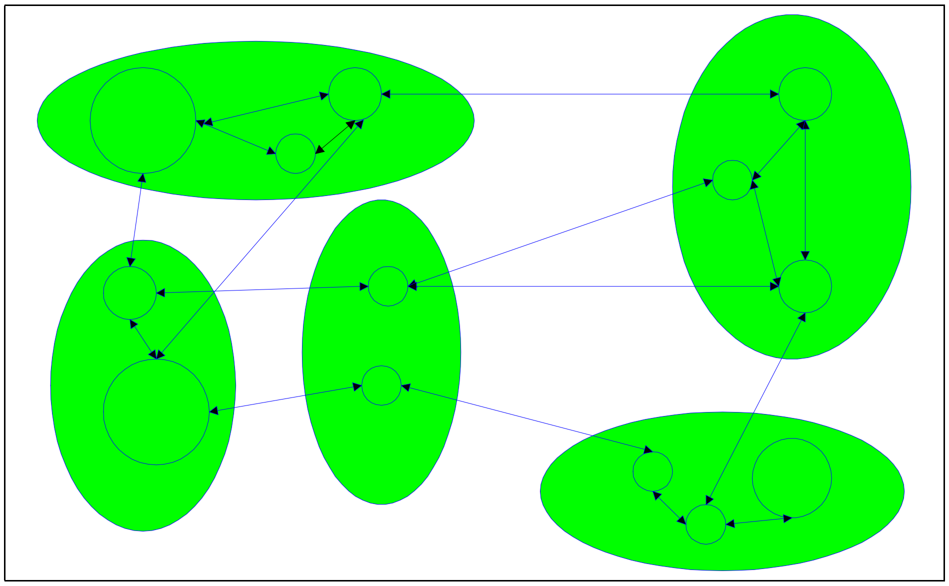

**Fig. 2.** Administrative Domains in the Internet

In Sect. 2 we review the work that has been done on the foundations of wide-area network programming languages. In Sect. 3 we review the support provided in traditional distributed programming languages for building fault-tolerant network applications. In Sect. 4 we give an informal overview of the ATF Calculus, intended as a kernel language for wide-area network programming languages with atomic failure as its central organizing principle. Finally Sect. 5 provides our conclusions.

## 2 Foundations of Wide-Area Languages

Wide-area programming languages are a relatively new area of research [9]. The central organizing principle of wide-area languages is *mobile computation*, in other words, the ability to transport computations between hosts in the network. Java applets are a popular example of this, allowing the code for Java classes to be downloaded across the network, with objects of these classes then created and invoked inside a Web browser [32]. Java servlets are a more ambitious example. In this case code is dynamically loaded into a server process. The application of this has been in building generic "application servers," with application-specific code loaded in response to client requests. More ambitious schemes envision client code uploaded to servers across the network [28].

A more general model of mobile computation is represented by moving state as well as code across the network. Several distributed programming systems, such as Aglets and Voyager [31, 39], support mobile objects, where the state of an object is serialized and transmitted between processes. At the receiving process, if the class of the serialized object is unknown, then the code for the class is dynamically obtained from the sender. This is similar to the notion of distributed closures popularized by the Obliq distributed scripting language [8]. Finally the Telescript language takes this idea one step further, allowing an entire thread state (including suspended procedure calls) to migrate between processes [50]. The suggested application for this functionality is in programming mobile agents that roam the network.

The formal foundations for such programming languages are beginning to emerge. Foundations for distributed programming languages in general have been proposed by actors [2], and by process calculi such as Communicating Sequential Processes (CSP) and the Calculus of Communicating Systems (CCS) [26, 35]. As an example of these, CCS is a simple "assembly language" for reasoning about concurrent computation. The syntax for CCS is described by:

$$P,Q ::= 0 \quad | \quad a().P \quad | \quad a\langle\rangle.P \quad | \quad (P \mid Q) \quad | \quad P+Q \quad | \quad P\setminus a$$

In this abstract syntax, 0 represents the stopped process. Processes communicate and synchronize by passing messages over channels. Channels are denoted by identifiers $a$. $a(\,).P$ represents a process that receives a message on the channel $a$ and then executes the code $P$. $a\langle\,\rangle.P$ represents a process that sends a message on the channel $a$. Communication is synchronous: the sending process blocks until another process executes the receive operation on that channel. In pure CCS, message-passing is only used for synchronization; there are no values in the calculus. This facilitates reasoning techniques that only consider the synchronization behaviour and evolution of processes (such as bisimulation and testing equivalence for verifying equivalence of processes, and temporal logic for reasoning about global properties of programs).

$(P \mid Q)$ denotes the parallel composition of two processes. $P+Q$ denotes choice; such a process can either commit to executing $P$ or $Q$. $P+Q$ may denote "external choice;" whether $P$ or $Q$ is chosen is determined by external processes sending or receiving messages, in the manner of the Ada select statement. $P+Q$ also denotes "internal choice," where a process may autonomously choose to execute $P$ or $Q$. Finally $P\setminus a$ denotes the "hiding" of any communications on the channel $a$ inside the process $P$.

Computation in CCS is given by a rule that allows a sending process to synchronize with the process that receives the message it is sending:

$$(a().P \mid a\langle\rangle.Q) \longrightarrow (P \mid Q)$$

This is a simplified computation rule, assuming that the synchronized processes are contiguous. Structural rules and congruential rules generalize this to the more general case where they may be embedded in other processes that are composed in parallel.

CCS is still very weak for reasoning about real programs. For example it is difficult to describe a simple client-server application where the client process sends a private reply channel to the server as part of its request message. The π-calculus is a major advance in this respect [36]. The π-calculus can be considered as building on CCS. There is one kind of value in the π-calculus: channel names. Channel names can be dynamically generated and transmitted between hosts. Furthermore the structural equivalence and computation rules of the π-calculus are carefully formulated to ensure that scoping of channel names is static. For example, transmitting a channel name between processes does not cause the name to be dynamically rebound to a different channel in the receiving process. This makes the π-calculus a very dynamic execution model, since the communication topology may be updated by the running processes, while the binding structure of the processes remains static. This latter fact makes the calculus amenable to static analysis and formal reasoning, just as static scoping of variable names is an important property of modern programming languages.

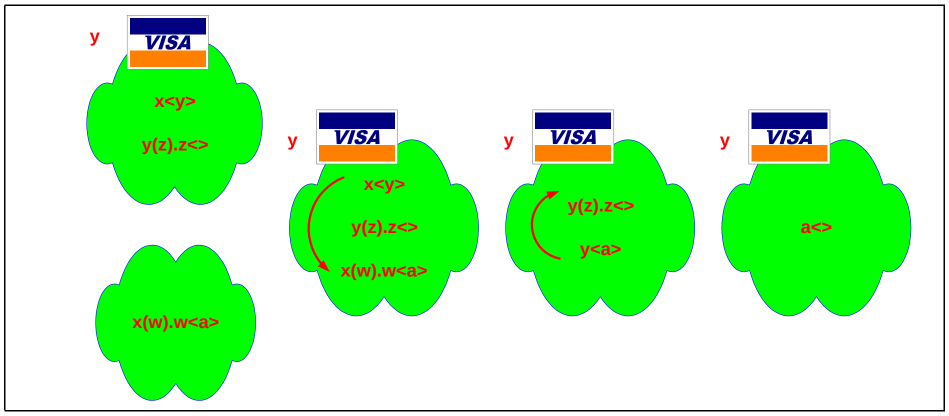

**Fig. 3.** Remote Procedure Call in π-calculus

The syntax for one variant of the π-calculus is given by:

$$P, Q ::= 0 \;\mid\; x(y).P \;\mid\; x\langle y\rangle \;\mid\; (P \mid Q) \;\mid\; (new\ x)P$$

This is a particular variant of the π-calculus, called the *asynchronous π-calculus* [27, 45]. This calculus is so named because the sending operation is non-blocking. The choice construct $P+Q$ is omitted because it is definable in the asynchronous π-calculus [27]. The scoping construct $(new\ x)P$ is different from the hiding construct $P \setminus a$ in CCS. The latter prevents any subprocess in $P$ from synchronizing with any outside process on the channel $a$; synchronization is still possible on other channels. The $P \setminus a$ construct does not prevent outside processes from synchronizing on the $a$ channel. This is not a problem for CCS, because of its static communication topology, but becomes an issue when channels or processes are transmitted in messages. For example CHOCS [48], a variant of CCS where processes are transmitted in messages, has dynamic scoping of channel names because of its use of the $P \setminus a$ construct. The $(new\ x)P$ construct on the other hand generates a new name $x$. If this name is transmitted in a message outside the process where it was created, the calculus guarantees that there will be no accidental name collisions. The computation rule for this calculus allows a process to receive a message that contains a channel name:

$$(x\langle y\rangle \mid x(z).P) \longrightarrow \{y/z\}P$$

where $\{y/z\}P$ denotes the capture-avoiding substitution of $y$ for free occurences of $z$ in $P$.

Fig. 3 gives an example of a remote procedure call in the π-calculus, where the client process sends a private reply channel $y$ to the server process in its request message:

$$\begin{aligned} CLIENT &= (new\ y)(x\langle y\rangle \mid y(z).z\langle\rangle) \\ SERVER &= x(w).w\langle a\rangle \end{aligned}$$

Here $x$ is the channel for request messages, and $a$ is the reply value returned by the server. The evolution of the client and server processes is described by:

$$\begin{aligned} ((new\ y)(x\langle y\rangle \mid y(z).z\langle\rangle)) \ \mid \ x(w).w\langle a\rangle \ &\equiv \ (new\ y)(x\langle y\rangle \mid y(z).z\langle\rangle \mid x(w).w\langle a\rangle) \\ &\longrightarrow (new\ y)(y(z).z\langle\rangle \mid y\langle a\rangle) \\ &\longrightarrow (new\ y)(a\langle\rangle \mid 0) \\ &\equiv \ a\langle\rangle \end{aligned}$$

There is an analogy between the private names generated in the π-calculus, and cryptographic keys for security protocols. This analogy is exploited in the Spi-calculus [1], an extension of the π-calculus with cryptographic primitives. The Spi-calculus can be used to give descriptions of cryptographic protocols, and proof techniques for process equivalence can be used to verify security properties based on verifying non-interference from "attacker" processes. Such proofs assume perfect cryptographic algorithms underlying the protocol descriptions. The use of statically scoped, dynamically generated names to model private keys is also used in the ambient calculus, described below.

Although models of concurrent computation, CCS and the π-calculus are not models of mobile computation. The higher-order π-calculus allows processes to be sent as the contents of messages [44]. However there is no distinction in this model between local and remote communication, and indeed this lack of distinction allows the higher-order π-calculus to be "compiled" to the π-calculus where only names can be transmitted. This compilation consists of passing the names of private "trigger channels," that the receiving process can then use to invoke the process that would be the contents of the message in the higher-order π-calculus.

More recent calculi have built on the example of the π-calculus to provide a foundation for wide-area languages where computation can migrate across the network. The *D*π-calculus (distributed π-calculus) includes a notion of *locations* [41]. Each process executes at a locality. A **go** instruction (inspired by the **go** operation in Telescript) migrates a running process from one location to another. The primary motivation for the

$D\pi$-calculus has been in reasoning about site failures, although it was subsequently adapted to reason about “untrusted” sites [42, 43]. The distributed join-calculus similarly has a notion of locations and site failures. The distributed join-calculus in addition has hierarchical nesting of locations [20, 21].

The most recent entry to this suite of calculi is the *ambient calculus* [11]. The ambient calculus has been designed from the start as a foundation for mobile computation in hierarchically nested administrative domains. In the philosophy of the ambient calculus, locality of processes is defined by the topology of the barriers partitioning the network into administrative domains. Process mobility is then described by a process crossing such a barrier (either entering or leaving an administrative domain). Although historically this model ties administrative domains to the network topology, an alternative interpretation is possible given the growing prevalence of virtual private networks (VPNs): the barriers comprise the administrative boundaries defined by VPNs, and barrier crossing corresponds to a process moving in and out of VPNs. In the ambient calculus, a barrier name comprises a capability for crossing the boundary, and security is based on the inability to cross a barrier. Thus a barrier name can be regarded as a private key for mediating access to a VPN.

An ambient has many interpretations. It could be a message, a communication channel, the address space of a heavyweight process, or a host on the network, or an administrative domain. For economy the calculus collapses all of these notions into the single notion of an ambient. In our work we find it necessary to distinguish these interpretations, leading to a loss of economy. An ambient in the ambient calculus is a collection of processes and subambients. Each ambient is named, and ambient names are the capabilities that mediate access to ambients. Every process executes in an ambient. Furthermore a process can only communicate with processes in that ambient. A process that wishes to communicate with a process in a different ambient must migrate to the remote ambient before initiating communication. There is no notion of channels, as in CCS and the $\pi$-calculus; channels are definable in terms of ambients [12].

The syntax of the ambient calculus is given by:

$$\begin{aligned} P,Q &::= 0 \mid M.P \mid n[P] \mid (P \mid Q) \mid (\mathit{new}\ n)P \\ P,Q &::= (x).P \mid \langle M \rangle \\ M &::= \mathit{in}\ n \mid \mathit{out}\ n \mid \mathit{open}\ n \mid M_1.M_2 \end{aligned}$$

Here $M$ denotes a sequence of capabilities. Capabilities allow a process to move into an ambient, or out of an ambient, or dissolve an ambient. $M.P$ denotes a process that executes the capability $M$ and becomes the process $P$. $n[P]$ denotes an ambient named $n$ at which the process $P$ is executing. $(\mathit{new}\ n)P$ denotes the dynamic generation of an ambient name. As in the $\pi$-calculus, although ambient names can be in messages that are exchanged between processes, the scoping of ambient names is guaranteed to be static. Messages are exchanged using the $(x).P$ and $\langle M \rangle$ constructs. The former describes a process that receives a message $x$ and then does $P$; there is no channel specified, since communication is local within an ambient. The latter $\langle M \rangle$ denotes a message (so again message-passing is asynchronous).

Fig. 4 gives an example of process migration in the ambient calculus. A thread executing in the ambient $p$ moves that ambient (and all of its executing threads $P$) out

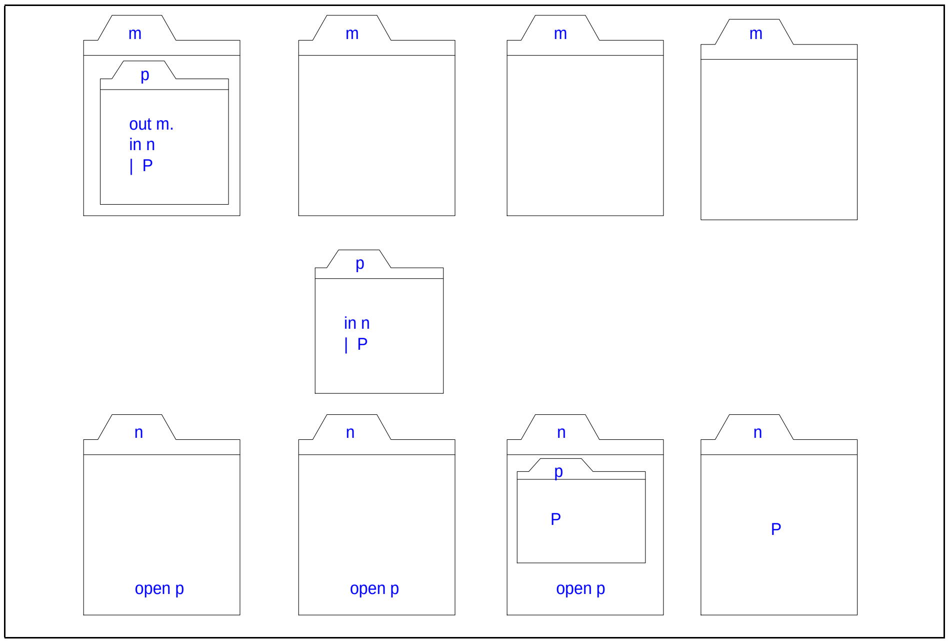

**Fig. 4.** Process Migration in Ambient Calculus

of the surrounding ambient *m* by executing the *out m* instruction. The ambient *p* is then moved inside the ambient *n* by executing the instruction *in n*. Finally a thread inside the ambient *n* dissolves the ambient *p* by executing the *open p* instruction. At this point the process *P* is ready to interact with any other processes in the ambient *n*. This evolution is described in the ambient calculus by:

$$\begin{aligned} m[p[\mathit{out}\ m.\mathit{in}\ n \mid P] \mid Q] \mid n[\mathit{open}\ p \mid R] &\longrightarrow m[Q] \mid p[\mathit{in}\ n \mid P] \mid n[\mathit{open}\ p \mid R] \\ &\longrightarrow m[Q] \mid n[p[P] \mid \mathit{open}\ p \mid R] \\ &\longrightarrow m[Q] \mid n[P \mid R] \end{aligned}$$

Security in wide-area network programming has become one of the central concerns in the ambient calculus. For example, to prevent the “leaking” of private keys by untrusted processes, an extension to the ambient calculus uses scoped key types to prevent leakage [10]. The Seal Calculus similarly incorporates mechanisms to prevent leaking of access rights in network programming [13].

## 3 Atomic Failure in Distributed Programming Languages

Achieving atomic failure is recognized as an essential step in building fault-tolerance into distributed applications. It is a form of hierarchical failure masking, providing a facility for rolling back system state to a consistent state in response to the failure of low-level operations. This liberates higher-level mechanisms to only have to consider the possibility of a collection of operations either succeeding or failing atomically. This layering of failure guarantees is in turn essential in handling the complexity of building fault-tolerant applications.

Various mechanisms have been proposed for achieving atomic failure. *Transactions* [30] have been used very successfully in databases to make compound database updates atomic. Atomicity here is used in two respects: an update either succeeds or fails atomically, and it is not possible to observe the interleaving of two updates. Transaction correctness is normally based on serializability, which requires that the schedule of executions of primitive update operations must be "equivalent" to a schedule where all of the components of an atomic update are scheduled continguously. "Equivalence" is normally based on being able to commute operations that are non-interfering. Another mechanism for achieving atomic failures is *reliable multicast* [29, 17]. Reliable multicast refers to the guarantee that, within a particular "group view," all processes in that group view receive a multicast or none of them receive it. Reliable multicast is typically provided with some delivery guarantees such as causal or total ordering on the delivery of the messages, and is used as a communication primitive for implementing fault-tolerant services based on replication. For example reliable multicast with causal ordering is the basic operation that is used in the state machine approach to building fault-tolerant distributed systems [46]. Reliable multicast is provided by various communication systems, including Isis, Horus, Psync and Transis [6, 7, 40, 4].

Transactions 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 [33], and subsequently in object-oriented distributed languages such as Avalon/C++ [18]. Transactions are an essential component in modern distributed programming environments such as the Object Management Group's Common Object Request Broker Architecture (OMG CORBA) and Microsoft Distributed Common Object Model (DCOM) [38, 16]. For example COM+ [14], the next generation of COM, is essentially the tight integration of COM with the Microsoft Transaction Service (MTS). A transaction service is also one of the basic mechanisms provided by the Jini infrastructure promoted by Sun Microsystems for programming network appliances [5]. Haines et al [23] describe a subroutine library in ML that enhances applications with transactional semantics. Their library provides undoability and persistence as orthogonal features, however they do not give a semantics for these features (while acknowledging that there is interaction between the features).

The transaction model adopted in distributed programming languages is that of *nested transactions* [34, 37]. This is motivated by considerations of nested remote procedure calls in client-server systems. For example, as depicted in Fig. 5, a request from Client A to Server B gives rise to further requests from Server B to Servers C and D. To make transactions scalable in a distributed environment where there is no central transaction manager, decisions on whether to commit or abort a transaction are performed locally. Each request gives rise to a new transaction. The original request from the client A gives rise to the root transaction at the server B, and a subsequent request from Server B to Server C gives rise to a new transaction at C's host that is a subtransaction of the transaction for B's service. Each subtransaction makes a tentative decision to commit or abort a transaction, and transmits this decision to the parent transaction. Eventually

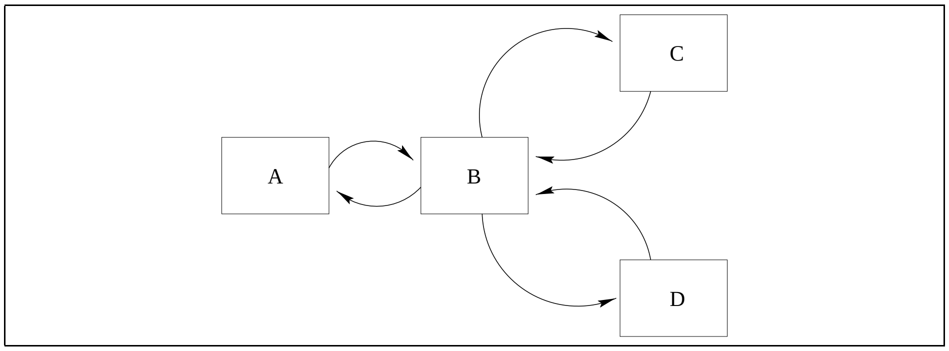

**Fig. 5.** Nested Remote Procedure Calls

these decisions are transmitted to the root transaction. This transaction then invokes a two-phase commit protocol, to atomically commit some subset of these transactions.

The semantics of nested transactions require that there be no interference between transactions, in order to preserve the serializability property of transactions. A child transaction inherits the locks obtained by its parent. To prevent interference, typically nested transaction systems distinguish between leaf transactions, which access data objects, and non-leaf transactions, which only invoke child transactions. Therefore a parent transaction never interferes with a child transaction accessing a data object. If the child transaction aborts, the parent transaction does not necessarily abort. The parent transaction may attempt the same request with a different service. On the other hand, if a parent transaction aborts then all of its child transactions must abort. So nested transactions incorporate a notion of *partial failures.*

In work on the foundations of wide-area languages, there has been limited consideration of failures. The $D\pi$-calculus and the distributed join calculus incorporate notions of locations and failures of locations [41, 21]. On the other hand, these approaches assume a fail-stop model of failures. Although this is reasonable for programming in local-area networks, it is an unrealistic assumption in wide-area networks. As already noted, wide-area networks essentially constitute asynchronous systems, where no bounds can be placed on message transmission delays or message processing times, or on the relative real-time clock drift between network hosts.

## 4 The ATF Calculus

The ATF Calculus is a new calculus, intended as the foundation of a wide-area network programming language, that takes *atomic failure* as its main organizing principle. This language builds on the work already done in wide-area programming languages, particularly the ambient calculus. However considerations of atomic failure lead to some fundamental rethinking of some of the design decisions in the ambient calculus. The motivation for atomic failure is that typical wide-area applications require some notion of coordination bettween geographically and administratively distributed sites. The distribution of the sites raises the possibility of localized site failures and network partitions, and there should be some support for coping with such failures. In an e-commerce application involving coordination between several sites, it should be possible to commit or abort a computation based on the pattern of failures of the sites involved in the computation.

Languages based on mobile computation essentially remove the problems with programming in wide-area networks by removing the network, i.e., a process moves to the site where it needs to communicate. While there is compelling motivation for this in real-time and interactive applications, this does not constitute the only class of applications that need to be programmed in wide-area networks. In particular, with the growing importance of business-to-business (B2B) e-commerce, there is a need for programming abstractions for building fault-tolerant applications over a federation of administrative domains. Formalisms such as the ambient calculus provide a foundation for designing programming languages for applications that span multiple administrative domains. Our objective is to develop the kernel of a programming language that incorporates ideas from the ambient calculus, while adding support for making failures atomic.

We call our kernel language the **ATF Calculus**. This calculus comes from adapting some of the concepts underlying existing foundations for mobile computation, principally the ambient calculus, and extending these concepts with support for atomic failures. This support is based on two ingredients:

– Tracking of causal dependencies between computations.
– An operation for atomic commitment of a collection of computations.

A reasonable design choice is to implicitly track the causal dependencies between actions. Such a causal dependency might arise for example between a write to a variable and a read from that variable. If the write operation must subsequently be undone, then the read operation and all subsequent operations in the thread that performed the read must also be undone. The difficulty with this approach is scalability: tracking causal dependencies at the level of primitive actions is much too low level to allow an efficient implementation. This also does not provide a way to specify that the failure of a collection of actions should be atomic.

We therefore introduce a new programming abstraction into an existing formalism for wide-area languages. We extend the ambient calculus with *transactions* (for want of a better word). A transaction is a collection of processes, whose actions succeed or fail atomically. As with nested transactions, a given transaction is local to a particular network host, so that logging, rollback and recovery only need to refer to the local log. On the other hand a transaction can fork subtransactions, with the obvious causal dependency arising between the parent transaction and the subtransactions.

The ATF Calculus semantics focuses on making failures atomic in a collection of cooperating processes. Concurrency control is regarded as an application issue that should be handled using known mechanisms (for example, locks and condition variables can be implemented using message-passing). Therefore, unlike nested transactions, the calculus allows interference between transactions, where one transaction can see the effects of another before the latter has committed those effects. This gives rise to a dynamically generated causal dependency from the latter to the former.

Causal cycles are possible, where several processes depend mutually recursively on each other's commitment. There is no scalable way to prevent such cycles from arising, so they must be accepted as a way of life. This raises the problem of somehow getting a collection of mutually dependent transactions to commit atomically. As already noted, non-blocking atomic commitment is known to be impossible in asynchronous systems [47, 24, 22]. We rely on a two-phase commit protocol to atomically commit a collection of (potentially mutually recursive) transactions.

In our framework, we do not tie request messages with corresponding reply messages. Although this has been the basis of the very successful use of remote procedure call in building LAN applications, it is not clear that this is the appropriate programming abstraction for building wide area applications. For example, remote procedure call hides several implicit messages that are exchanged "under the covers," including request and reply acknowledgements, retransmissions of unacknowledged messages, and periodic "pings" of the server by the client. In the setting of federated administrative domains, these may be useful abstractions that we wish to build on top of the basic functionality. However the kernel of our calculus is asynchronous message-passing where messages may need to be delivered across administrative domains.

Because of this decision, we must rely on the application to deliver messages across the boundaries of administrative domains. For example some of these applications may be authentication servers. Certainly there is a need to distinguish "system" messages that do not give rise to new causal dependencies (we do not want all applications to become causally dependent on an authentication server). More fundamentally we must be wary of for example malicious administrator processes that attempt to get the system into an inconsistent state. Consistency is based on the idea that there is an "abstraction mapping" from an execution trace of the system to a trace where no failures have happened. Therefore correctness is based on the notion of *causal consistency*: this means that in the trace resulting from the abstraction mapping, no committed process uses the output of an aborted process.

## 4.1 Network Model

Traditional process calculi such as CCS and the π-calculus do not distinguish between processes and the medium through which they communicate; the medium is represented as just another process. The ambient calculus does not distinguish between networks, hosts and process address spaces; all are represented as ambients. We find it useful to distinguish between places on the network (hosts and subnets) and processes, because of their different failure characteristics. In this section we describe our model of networks. In the next section we describe the processes that execute over these networks.

Networks in the ATF Calculus are described as follows:

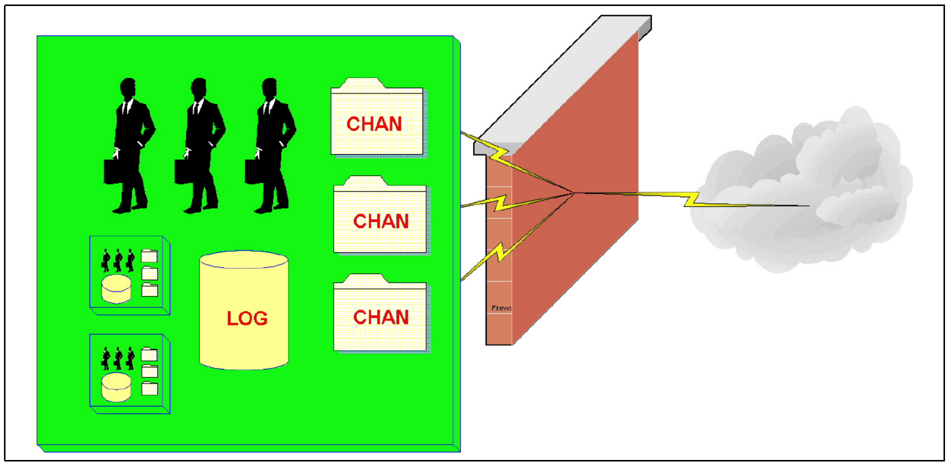

**Fig. 6.** Structure of a Place in the ATF Calculus

$$
\begin{array}{rcll}
n \in \text{Name} & ::= & n^{\text{pl}} & \text{Place Name} \\
 & \mid & n^{\text{ch}} & \text{Channel Name} \\
 & \mid & n^{\text{pk}} & \text{Packet Name} \\
 & \mid & t & \text{Transaction Name} \\
N \in \text{Network} & ::= & n^{\text{pl}}[\mathcal{L}, P] & \text{Place} \\
 & \mid & t\{P\} & \text{Process} \\
 & \mid & (N_1 \mid N_2) & \text{Wire} \\
 & \mid & (new\, n : NT)N & \text{New Name} \\
 & \mid & \mathbf{0} & \text{Empty Network} \\
\mathcal{L} \in \text{Log} & ::= & \varepsilon & \text{Empty Log} \\
 & \mid & N & \text{Log Entry} \\
 & \mid & \mathcal{L}_1 ; \mathcal{L}_2 & \text{Log Extension}
\end{array}
$$

A place is an administrative domain or a host on the network. This contains a “soup” of processes, channels and subdomains $P$. $n^{\text{pl}}$ is the unique name of the place, while $\mathcal{L}$ is a log of the actions performed at that place. For fault tolerance this log must be kept in stable storage.

A channel is used for asynchronous communication between processes. As in the Ambient Calculus and the Seal Calculus, all communication is local within an administrative domain. To communicate with a remote host, there must be an application-level protocol for delivering a message across the intervening domains. As with places, a channel has a name $n^{\text{ch}}$. The contents of a channel $n^{\text{ch}}$ at a place $n^{\text{pl}}$ is obtained by taking the union of atoms of the form $t\{send(in\, n^{\mathrm{ch}}, V)\}$. Each such atom denotes a single-element channel containing a *transactional value* $t\{V\}$, a value generated by a process executing as part of the transaction $t$.

A *transactional process* $t\{P\}$ denotes a process $P$ that executes as part of the transaction identified by the transaction identifier $t$. If that transaction is aborted, then all effects (messages sent and received) of that process must be undone. Transactional values $t\{V\}$ and transactional processes $t\{P\}$ reflect the fact that all application-level computations in the ATF Calculus take place within a transaction. The transaction name $t$ identifies the transaction within which a process executes ($t\{P\}$) or within which a value has been computed ($t\{V\}$).

## 4.2 Transactional Processes

Processes in the ATF Calculus are described by:

$$\begin{array}{rcll}
P \in \text{Process} & ::= & send(M,V) & \text{Message Send} \\
& | & receive(M,F_1,F_2) & \text{Message Receive} \\
& | & crypt(M,V,F) & \text{Packet Encrypt} \\
& | & decrypt(M,V,F) & \text{Packet Decrypt} \\
& | & F(V) & \text{Application} \\
& | & let\ \langle \overline{x_n} \rangle = V\ in\ P & \text{Elim Tuple} \\
& | & !P & \text{Replication} \\
& | & (P_1 \mid P_2) & \text{Parallel Composition} \\
& | & (new\, n : NT)P & \text{New Name} \\
& | & commit & \text{Commit transaction} \\
& | & abort & \text{Abort transaction} \\
& | & \mathbf{0} & \text{Stopped Process} \\
F \in \text{Cont} & ::= & (x : T)P & \text{Continuation}
\end{array}$$

The basic operations are for asynchronous message-passing [45]. In the ATF Calculus the essential use of mobility for navigating administrative domains is in the *send* and *receive* operations. For example the *send* operation takes two arguments: a capability $M$ and a value $V$. The capability specifies a path to be taken in the network (identified by subcapabilities for leaving and entering administrative domains) and a capability for depositing the value in a channel at the final destination place. As with algebras such as the Ambient Calculus and the Spi Calculus [11, 1], access control is enforced by controlling the distribution of these capabilities, which are akin to private keys in cryptographic infrastructures.

Our provision for “mobility” is consistent with approaches in active networks, such as the Switchware architecture [3], that restrict mobile threads to simplified packet languages (such as the PLAN language of the Switchware architecture [25]). The language of capabilities $M$ can be considered as the analogue in the ATF Calculus of packet languages such as PLAN. This is in contrast with the Ambient Calculus and the SEAL Calculus, which allow general user processes to migrate across the network.

The *receive* operation takes as its main argument a capability for reading from a channel. This operation also has two continuations: the success continuation $F_1$ and the failure continuation $F_2$, where the latter is invoked if the receive operation times out. We assume an asynchronous system, where no upper bound can be placed on message delivery delays or relative clock drift, so timeouts provide a form of unreliable failure detector which is the best that we can attain in an asynchronous system.

The constructs for parallel composition $(P_1 \mid P_2)$, generation of new (channel and packet) names $(\mathit{new}\ n : NT)P$ and stopped processes $\mathbf{0}$ are syntactically similar to network wires $(N_1 \mid N_2)$, generation of new (place) names $(\mathit{new}\ n : NT)N$ and empty networks $\mathbf{0}$. The semantic difference is that the former are used to build descriptions of processes that execute within a transaction. This means that there is a degree of "impermanence" to the former, since processes evolve during execution. In our calculus there is also a degree of "tentativeness" to process descriptions since the abortion of a transaction will require the undoing of all processes that are part of that transaction or may depend causally on that transaction. On the other hand we assume (for now) a static network topology, because of the need to maintain places as "permanent" entities with an associated log in stable storage.

Values in the ATF Calculus are described by:

$$\begin{array}{rcll}
M, V \in \text{Value} & ::= & n^{\text{pk}}[V] & \text{Packet} \\
& \mid & \langle V_1, \ldots, V_k \rangle & \text{Tuple} \\
& \mid & p & \text{Parameter} \\
& \mid & \mathit{in}\ p & \text{Input Capability} \\
& \mid & \mathit{out}\ p & \text{Output Capability} \\
& \mid & M_1.M_2 & \text{Compose Caps}
\end{array}$$

Packets are an inessential aspect of the language, but provide cryptographic primitives for authentication and encryption of message contents as part of the language. Creating a packet requires a capability for putting a value into a packet, while reading a packet requires the inverse capability. Capabilities therefore provide a function analogous to cryptographic keys; an application may publish only a key for creating packets, and then be the only process capable of reading packets created using the capability. Separating the capabilities for message creation and reading, from the capabilities for delivery and receipt of messages, allows an encrypted message to be forwarded by a process that does not have access to a key for reading the message contents.

The form of capabilities are taken from the Ambient Calculus. In the Ambient Calculus, places, channels and packets are uniformly represented as ambients. However we find it easier to treat each of these concepts differently. For fault tolerance purposes, the behaviour of places, channels and packets are very different. A capability is a sequence of subcapabilities of the form *in p* or *out p*, where $p$ is a place name, channel name or packet name, with the interpretation:

| | |
|---|---|
| $\mathit{in}\ n^{\mathrm{pl}}$ | enter a place |
| $\mathit{in}\ n^{\mathrm{ch}}$ | write to a channel |
| $\mathit{in}\ n^{\mathrm{pk}}$ | create a packet |

| | |
|---|---|
| $\mathit{out}\ n^{\mathrm{pl}}$ | leave a place |
| $\mathit{out}\ n^{\mathrm{ch}}$ | read a channel |
| $\mathit{out}\ n^{\mathrm{pk}}$ | read a packet |

Useful capabilities have the following forms:

| | |
|---|---|
| $\mathit{in}\ n^{\mathrm{pk}}, \mathit{out}\ n^{\mathrm{pk}}$ | Create or destruct a packet |
| $\mathit{out}\ n^{\mathrm{ch}}$ | Receive from local channel |
| $M_1 \ldots M_n.\mathit{in}\ n^{\mathrm{ch}}$ | Deliver payload to channel at destination |

where in the latter case each $M_i$ has the form $\mathit{in}\ n_i^{\mathrm{pl}}$ or $\mathit{out}\ n_i^{\mathrm{pl}}$ for some $n_i^{\mathrm{pl}}$, $i = 1, \ldots, n$.

Types in the ATF Calculus are described by:

$$\begin{array}{rcll}
\mathit{AT} \in \text{Ambient Type} & ::= & \mathit{Place} & \text{Place Type} \\
& \mid & \mathit{Chan}[T] & \text{Channel Type} \\
& \mid & \mathit{Packet}[T] & \text{Packet Type} \\
\mathit{NT} \in \text{Name Type} & ::= & \mathit{AT} & \text{Ambient Type} \\
& \mid & \mathit{Trans} & \text{Transaction Type} \\
T \in \text{Type} & ::= & \mathit{NT} & \text{Name Type} \\
& \mid & \langle T_1, \ldots, T_k \rangle & \text{Tuple Type} \\
& \mid & \mathit{Cap}[\mathit{AT}] & \text{Capability Type}
\end{array}$$

Ambient types are the types of place names, channel names and packet names (the name reflects the original inspiration from the Ambient Calculus). Types also include process types and capability types, where the latter are indexed by ambient types.

## 5 Conclusions

We have provided an overview of the ATF Calculus, a kernel language for wide-area network programming languages with support for making failures atomic in network programming. This language builds on work in calculi for mobile computation in WAN programming, and on work on transactional mechanisms for LAN programming languages.

An interesting question is what process equivalences can be defined. The complication is that the liveness properties of processes may be quite different unless one also considers uncommitted effects. However considering such effects introduces the possibilities of negative effects (undoing message sends due to aborts) and it is not clear how to obtain a fixed point. This remains an interesting topic for future work.

It is straightforward to add an operation to the ATF Calculus for creating new transactions (with child transactions depending causally on parent transactions). We are investigating this issue in the context of considering more flexible ways for creating and combining transactions.

## 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. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed SYstems*. MIT Press, 1988.
3. D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The Switchware active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, 1998.
4. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84. IEEE Computer Society, 1992.
5. K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.
6. K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.
7. K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.
8. Luca Cardelli. A language with distributed scope. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 286–297, San Francisco, California, January 1995. ACM Press.
9. Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
10. Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. To appear, 2000.
11. 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.
12. Luca Cardelli and Andrew Gordon. Types for mobile ambients. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Antonio, January 1999. ACM Press.
13. Guiseppe Castagna and Jan Vitek. A calculus of secure mobile computations. In *Internet Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
14. David Chappell. COM+: The next generation. *Byte Magazine*, December 1997.
15. William Cheswick and Steven Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.
16. Microsoft Corporation. DCOM technical overview. Technical report, Microsoft Corporation, 1996.
17. F. Cristian, H. Aghili, H. R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, pages 200–206, 1985. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).
18. D. Detlefs, M. Herlihy, and J. Wing. Inheritance of synchronization and recovery properties in avalon/c++. *IEEE Computer*, pages 57–69, December 1988.
19. 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.
20. 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.

21. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag. LNCS 1119.
22. 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.
23. N. Haines, D. Kindred, J. G. Morrisett, and S. M. Nettles. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, November 1994.
24. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.
25. Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of ACM International Conference on Functional Programming*. ACM Press, September 1998.
26. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
27. 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.
28. Dag Johanssen. Mobile agent applicability. In *Mobile Agents Workshop*, Stuttgart, Germany, September 1998.
29. L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.
30. Butler Lampson. Atomic transactions. In B. Lampson, M. Paul, and H. Siegert, editors, *Distributed Systems–Architecture and Implementation*, volume 205 of *Lecture Notes in Computer Science*, pages 246–285. Springer-Verlag, 1981.
31. Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
32. Sheng Liang. Dynamic class loading in the Java virtual machine. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*. ACM Press, October 1998.
33. Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
34. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufman, 1994.
35. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
36. Robin Milner. The polyadic $\pi$-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.
37. J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
38. Object Management Group. *The Common Object Request Broker: Architecture and Specification 2.0*, July 1995.
39. Objectspace Inc. *Overview of Voyager: ObjectSpace's Family for State-of-the-Art Distributed Computing*, 1999.
40. L. L. Peterson, N. C. Bucholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, 1989.
41. James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proceedings of the International Conference on Automata, Languages and Programming*, 1997.
42. James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

43. James Riely and Matthew Hennessy. 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.
44. Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. CST-99-93.
45. Davide Sangiorgi. Asynchronous process calculi: The first-order and higher-order paradigms. *Theoretical Computer Science*, 1999.
46. Fred Schneider. Implementing fault-tolerant services using the state-machine approach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.
47. Dale Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California, Berkeley, 1982.
48. B. Thomsen. A calculus of higher order communicating systems. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 143–154. ACM Press, January 1989.
49. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*, Lecture Notes in Computer Science, pages 49–64. Springer-Verlag, 1997.
50. James E. White. Telescript technology: The foundation for the electronic marketplace. Technical report, General Magic White Paper, 1994.