Electronic Notes in Theoretical Computer Science 68 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 19 pages

Toward a Programming Model for Building Reliable Systems with Distributed State

John Field

IBM T.J. Watson Research Center
jfield@watson.ibm.com

Carlos Varela

Department of Computer Science
Rensselaer Polytechnic Institute
cvarela@cs.rpi.edu

Abstract

We present the preliminary design of a programming model for building reliable systems with distributed state from collections of potentially unreliable components. Our *transactor* model provides constructs for maintaining *consistency* among the states of distributed components. Our intention is that transactors should support key aspects of both traditional distributed transactions, e.g., for electronic commerce, and systems with weaker consistency requirements, e.g., peer-to-peer file- and process-sharing systems. In this paper, we motivate the need for language support for maintenance of distributed state, describe the design goals for the transactor model, provide an operational semantics for a simple transactor calculus, and provide several examples of applications of the transactor model in a higher-level language.

1 Introduction

Many distributed systems must maintain *distributed state*. By this, we mean that the states of several distributed components in a network-connected system are interdependent on one another. The classical example of such a scenario is a bank transaction involving the transfer of money from one account to another, where we must ensure that it is not possible (even in the presence of a system failure) for one account to be debited without a corresponding credit being made to the other account, and vice-versa.

Ensuring that these interrelated states are maintained in a *consistent* way in a wide-area network—where transmission latencies may be high, and where node and link failures are relatively common occurrences—is difficult. Traditionally, distributed state maintenance has been viewed primarily as a systems or "middleware" [5] problem, in which, e.g., system infrastructure for message-passing provides guaranteed message delivery on an unreliable network substrate [6,25], or where distributed databases or transaction systems support the illusion of shared, atomically-updatable state across multiple nodes [17,22]. However, in an open and heterogeneous world—where software components are designed and implemented independently, and where they connect to one another fluidly—it is unrealistic to assume the existence of common system infrastructure for distributed state maintenance. While a system-neutral API for distributed transaction management exists [21], it provides support only for a single, rigid consistency model.

Consider a collection of distributed components that engage in a protracted negotiation toward some mutually-desirable outcome (e.g., an auction). The negotiation process will entail sending messages among the components, and updating each component's state in various ways during that time. If the negotiation is successful, that subset of the components that have reached an agreement will want to ensure that each of their states consistently reflects the agreed upon outcome; if the negotiation fails, there is typically no need to ensure that the states are mutually consistent at the end of the negotiation process, provided that each participant has reached a satisfactory local state.

Distributed transaction management systems typically require that all of the participants in the transaction coordinate their work with a pre-designated *transaction manager*, and that every transaction has well-defined beginning and end points. These properties make it difficult to build open distributed systems where the topology of the system is determined dynamically, where the scope of—and even the need for—a transaction is situation-dependent, and where transactional and non-transactional components can easily interact.

In this paper, we present preliminary work on what we will call the *transactor* programming model. The transactor model allows separately-developed distributed components to be dynamically combined, but supports maintenance of consistent distributed state *for those components that require it*. The transactor model is based on the *actor* model introduced by Hewitt [15], and further refined and developed by Agha et al. [4,3,24]. Actors are inherently independent, concurrent, and autonomous which enables efficiency in parallel execution [16] and facilitates mobility [2,1]. The actor model and languages provide a very useful framework for understanding and implementing open distributed systems.

Transactors can be regarded as a *coordination model* [12,13,26,8,14,9], in the sense that they are intended primarily to express the semantics of the *interactions* among various distributed components, rather than to describe the computations local to a node in the system.

1.1 Design Desiderata

The design desiderata for the transactor model are as follows:

- The model should allow arbitrary collections of concurrent processes, which may be interconnected in a dynamically-updatable topology.
- The model should expose the possibility of network link and node failures to the programmer, and thus allow the component's responsibilities and guarantees in the presence of failure to be made *explicit*.
- The model should not require an omniscient central coordinating entity to implement.
- Communication should be based on message-passing, not shared memory.
- The model should incorporate explicit support for stable state checkpoints and rollback, to allow computations that have become inconsistent, have failed, or have resulted in a runtime error to recover in a consistent state.
- The model should incorporate a mechanism for discovering and reacting to state inconsistency.

An important transactor design principle is to avoid *requiring* that any component of a system implement more than a minimal set of primitives needed to allow composite systems to be built at all. For example, we would like to avoid requiring that every component of a composite system necessarily be able to participate in a distributed 2-phase commit protocol, yet we would like to be able to take advantage of components that provide such guarantees. Moreover, we wish to be able to combine both high- and low-reliability components, reason about the behavior of the composite system, and supply additional software layers to improve its reliability if desired.

By exposing key semantic concepts related to maintenance of distributed state in a common, well-founded *language*, rather than relegating these issues to system or middleware, composite distributed applications can reason about the failure semantics of their components, and, if appropriate, supply extra protocol layers (e.g., logging, rollbacks, retries, replication, etc.) to add additional reliability. Use of a common interconnection language also facilitates testing, porting, and simulating of internet-scale software, something that is currently extremely difficult to do without deploying a full-blown production system.

1.2 Related Work

While there is much existing foundational work on languages for concurrent, and to a lesser extent, distributed systems (e.g., actors [4,3], the π-calculus [18], the join calculus [11], and mobile ambients [7]), we are not aware of formalisms that provide primitives for reasoning about the consistency of distributed state in the presence of failures. At the other end of the spectrum, distribution in "industrial" languages or language models, e.g., Java RMI [23] and Jini [27], CORBA [19], and COM+, is generally based on remote procedure call models that have limited mechanisms for dealing with failure, and are best suited for tightly-coupled, centrally-managed applications.

Liskov's Argus language [17], is the work closest in spirit to ours. It incorporates constructs for maintenance of distributed state (via *nested transactions*). Liskov introduced two principal abstractions: guardians and actions. A guardian is an abstract object whose purpose is to encapsulate a resource or resources. Special procedures, called handlers, can be used to access a guardian. An action is essentially a nested atomic transaction. Argus provides a programming interface onto centrally-managed nested transactions. By contrast, with transactors, we intend to uniformly model a variety of failure-management techniques, including transactions and applications with weaker consistency semantics.

1.3 Outline

The remainder of this paper is organized as follows: First, Section 2 gives a high-level description of the transactor model, and illustrates the model using a simple persistent `Counter` example with synchronized access. Next, Section 3 provides an operational semantics for a lambda-based functional transactor language. Following, Section 4 describes an electronic commerce example involving a distributed transaction. Finally, we conclude with a discussion of open questions and future work.

2 The Transactor Model

Transactors are defined using the following core primitives and assumptions:

- Transactors can respond to asynchronous messages by creating new transactors, sending messages to other known transactors, or changing their internal state (these are the core concepts of the actor model [4]).
- Messages are not guaranteed to arrive to target transactors. However, if a message arrives, it does not arrive corrupt.
- A transactor may decide to *commit* its current state to a *stable* state. When a transactor becomes stable, its state will not change in future communications. The transactor may fail to respond to messages if its node is down, or the network is partitioned; however, when the node comes back up, or the network gets reconnected, the transactor will appear at its stable state.
- When an unstable transactor decides to *roll-back*, or is required to roll-back by its run-time system, new messages from that transactor will trigger roll-back behavior for transactors whose state depended on the state of the unstable transactor.

An important characteristic of our reasoning framework is its layered architecture. We do **not** assume that the network is reliable. That is, we do not

```
// Counter:
//
// Implements synchronized and persistent access to a
// counter value as a chain of stable (committed)
// transactors.  All read or write requests are
// forwarded to the last stable transactor in the chain.
// All writes are required to be ‘‘stabilizable’’, in the
// sense that the sender of the write request must
// itself be stable, thus assuring permanence of the
// written value.  A write request by an unstable
// transactor will fail.
//
transactor Counter(int init_value) {

  int current_value = init_value;

  // non-Null if this counter is stable (i.e., has been
  // committed); end of chain of transactors rooted at
  // next_val yields last committed value
  Counter next_val = Null;

  // read(requester):
  //
  // sends a message to requester with latest stable
  // (committed) value for counter
  //
  read(CounterReader requester) {
  if ( volatility == stable ) then
    // this value is stable (committed);
    // see if other committed values exist
    next_val.read(requester);
  else
    // first non-committed value, which (by convention)
    // must be equal to last committed value
    requester.returnedVal(current_value);
  }

  // incr():
  //
  // increments latest stable (committed) value of
  // counter by incr_value and attempts to commit by
  // stabilizing; stabilization will only succeed if
  // sender is itself stable; otherwise, the transactor
  // will roll back.
  //
  incr(int incr_value) {
    if ( volatility == stable ) then
      // this value has been committed - can’t update;
      // find uncommitted value in chain
      next_val.incr(incr_value)
    else {
      // this value is uncommitted; attempt to update
      // and commit
      current_value = current_value + incr_value;

      // spawn new transactor to handle subsequent
      // requests
      next_val = new Counter(current_value);

      // attempt to stabilize; if stabilization fails
      // because sender is unstable, rollback to
      // previous value
      stabilize;
      if ( volatility != stable )
        rollback;
    }
  }
```

Fig. 1. A simple transactor implementing synchronized, persistent access to a counter.

assume guaranteed message delivery, or synchronous channel name passing. Libraries that provide stronger semantic guarantees that the basic transactor model can be defined if required.

The goal is to provide a completely decentralized and distributed weak consistency protocol. Applications requiring stronger semantic properties will trigger validation algorithms to reach the desired consistency semantic guarantees. Locally persistent state is provided as a primitive to transactors wishing to *stabilize*. Roll-back behavior is also supported by persistent intermediate checkpoints, transparent to applications.

The transactor model admits a number of programming language realizations. In examples in the sequel, we will use a realization that has an object-based flavor.

2.1 A Transactor Example

Figure 1 depicts the definition of `Counter`, a very simple transactor. While this example does not illustrate the full power of the transactor model in a distributed setting, it does cover several of the ideas underlying transactors. The `Counter` transactor implements synchronized and persistent access to a counter value. It is synchronized, in the sense that the counter is incremented atomically. It is persistent, in the sense that readers of the counter will never read a value that may be subsequently rolled back or corrupted by failure, although the counter may become inaccessible due to network failures, and other writers may subsequently update it.

Like conventional objects, the `Counter` transactor has fields representing the current state of the transactor. However, unlike regular objects, access is implicitly protected and synchronized:

- a transactor's state may only be accessed via a *message processor* (which resembles a method, but is invoked asynchronously and therfore does not explicitly "return").
- only one message processor may be active at a time, and must complete before another message may be processed.
- multiple message processor invocations (i.e., "message sends") on the same transactor are implicitly queued, and processed one at a time.

Since message processors are not methods, a message processor such as `read` that is intended to return a value must take as an argument a transactor (which here is deemed to have type `CounterReader`) to receive the returned value. Message sending is *asynchronous*, that is, the sender does not wait for any acknowledgment before continuing processing. Furthermore, there is no guarantee of message delivery. Both synchronous invocation and guaranteed delivery can be provided as higher-level abstractions.

The most notable aspect of `Counter` is the fact that it maintains a *chain* of *stable* (i.e., committed) transactor values, rather than a single value. While this may seem to be an egregious waste for such a simple example, it illustrates a general transactor programming principle: each distinct transactional "unit of work" (here, a single invocation of the `incr` message processor) corresponds to a distinct transactor.

The `stabilize` keyword in the `incr` message processor requires some elaboration: this construct "commits" a transactor's state by ensuring that each *sender* of a message to the transactor that has *updated the transactor's state* (in the `Counter` example, this would be any invocation of the `incr` message processor) is itself stable. Once a transactor is stable, attempts to invoke any message processor that can mutate the transactor's state are ignored. In general, an attempt to stabilize a transactor may fail, since it requires that the set of all senders of mutating messages must also be stable. In the case of `Counter`, failure to stabilize results in rollback of the transactor. In general, such rollbacks may trigger rollbacks in other transactors with which a transactor has communicated. In the case of `Counter`, however, such cascading rollbacks cannot occur.

In order to support stabilization of mutually dependent transactors, we introduce a `quiesce` primitive. A transactor in the *quiescent* state can still send and receive messages, but it makes a promise not to change its state unless it is forcibly rolled back due to receipt of a message from a partner transactor with which it had previously communicated, but which has since rolled back transactor. The quiescent state is persistent, in the sense that it can be recovered after a temporary failure such as a hardware reboot.

3 Toward a Formal Operational Semantics

The example in 2.1 was written in a loosely defined language with an object-oriented flavor. To make the concepts of the transactor model precise, we modify the formal semantics of actors formulated by Agha, Mason, Smith and Talcott [3]. The following two subsections introduce a transactor calculus and its operational semantics. In this paper, we will not define a formal translation from the high-level language used in the examples to the lower-level calculus; doing so would be tedious but not difficult.

3.1 A Simple Lambda Based Transactor Calculus

Our transactor calculus is a simple extension of the call-by-value lambda calculus that includes—in addition to arithmetic primitives and structure constructors, recognizers, and destructors—primitives for creating and manipulating transactors. A transactor's behavior is described by a closure which embodies the code to be executed when a message is received. In general, this closure will be computed anew for each message received, and thus embodies the current *state* of the transactor. The transactor primitives are:

`new`(v) creates a new transactor with behavior v and returns its name.

`send`(t, v) creates a new message with receiver t and contents v and passes it to the message delivery system.

`ready`(v) indicates that the transactor has completed processing the current message, and is ready to process the next message with behavior v.

`quiesce`(e) causes the transactor to enter the *quiescent* volatility state, in which all future messages are processed with "immutable", behavior e. A quiescent transactor may however still roll back due to dependencies on other transactors. This state is similar to the first phase in a two-phase commit protocol.

`stabilize`() attempts to change the transactor's volatility state from *quiescent* to *stable*; a stable transactor not only has "immutable" behavior, but will never roll back. This transition is only successful if all transactors on which the current transactor is dependent are themselves stable. The primitive yields the value `true` if the transition is successful, `nil` otherwise.

`rollback`() rolls the transactor back to its initial behavior.

`volatility`() returns the transactor's volatility state: *volatile*, *quiescent*, or *stable*.

3.2 Operational Semantics

We give the semantics of transactor expressions by defining a transition relation on configurations—global snapshots of a set of transactors. We first define values, expressions, messages, volatility values, dependence maps, and stability states. Then, we define a set of operations on transactor dependence maps. Finally, we define configurations and the single-step transition relation among configurations.

Let $\mathbf{M}_\omega[\mathsf{M}]$ be the set of (finite) multi-sets with elements in M, $\mathsf{X}_0 \xrightarrow{f} \mathsf{X}_1$ be the set of partial finite maps from X_0 to X_1, and Dom(f) be the domain of f. For any function f, $f\{\mathrm{x} \to \mathrm{x}'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{\mathrm{x}\}$, $f'(\mathrm{y}){=}f(\mathrm{y})$ for $\mathrm{y}{\neq}\mathrm{x}$, $\mathrm{y} \in \mathrm{Dom}(f)$, and $f'(\mathrm{x}){=}\mathrm{x}'$. Let $\emptyset$, where appropriate, be the function f such that $\mathrm{Dom}(f) = \emptyset$, $\{\mathrm{x} \to \mathrm{x}'\}$ be $\emptyset\{\mathrm{x} \to \mathrm{x}'\}$, and $f\{\mathrm{x} \to \mathrm{x}', \mathrm{y} \to \mathrm{y}'\}$ be $f\{\mathrm{x} \to \mathrm{x}'\}\{\mathrm{y} \to \mathrm{y}'\}$.

3.2.1 Values, Expressions, Messages, Volatility Values, Dependence Maps, and Stability States

We take as given countable sets At (atoms), X (variables), and N (natural numbers). We assume At contains *true*, *nil* for booleans, $\mathcal{V}$, $\mathcal{Q}$, and $\mathcal{S}$ for volatility values, as well as integers. F_n is the set of primitive operations of rank n, which includes arithmetic operations, branching, pairing and transactor primitives `new`, `send`, `ready`, `quiesce`, `stabilize`, `rollback`, and `volatility` (ranks 1,2,1,1,0,0, and 0).

Definition $(\mathsf{V}, \mathsf{E}, \mathsf{M}, \mathsf{W}, \mathsf{D}, \mathsf{S})$**:** The set of *values*, V, the set of *expressions*, E, the set of *messages*, M, the set of *volatility values*, W, the set of *dependence maps*, D, and the set of *stability states*, S, are defined inductively as follows:

$$\mathsf{V} = \mathsf{At} \cup \mathsf{X} \cup \mathsf{W} \cup \lambda\mathsf{X}.\mathsf{E} \cup \mathtt{pr}(\mathsf{V}, \mathsf{V})$$
$$\mathsf{E} = \mathsf{V} \cup \mathtt{app}(\mathsf{E}, \mathsf{E}) \cup \mathsf{F}_n(\mathsf{E}^n)$$
$$\mathsf{M} = {<}\mathsf{X} \Leftarrow \mathsf{V}{>}_\mathsf{D}$$
$$\mathsf{W} = \{\mathcal{V}, \mathcal{Q}, \mathcal{S}\}$$
$$\mathsf{D} = \mathsf{X} \xrightarrow{f} \langle\, \mathsf{W}, \mathsf{N} \,\rangle$$
$$\mathsf{S} = \langle\, \mathsf{W}, \mathsf{N}, \mathsf{E}, \mathsf{E}, \mathsf{D}, \mathsf{D} \,\rangle$$

We use variables for transactor names. A transactor can be either ready to accept a message, written $\mathtt{ready}(v)$, where v is a lambda abstraction denoting its behavior; or busy executing an expression, written e. A message to a transactor with name t, contents v, and dependence map δ, is written ${<}t \Leftarrow v{>}_\delta$. We let w range over $\{\mathcal{V}, \mathcal{Q}, \mathcal{S}\}$ for transactor volatility values, representing *volatile*, *quiescent*, and *stable*, respectively. It will be convenient to assume that volatility values are ordered by $\mathcal{V} < \mathcal{Q} < \mathcal{S}$. We use natural numbers for a transactor *incarnation*—the number of times the transactor has rolled-back. A dependence map specifies the dependencies for a given transactor: for each transactor that it is dependent on, it maps the name, t, into $\langle\, w, i \,\rangle$, which contains its last-known volatility value, w, and its last-known incarnation value, i. A transactor's stability state, $\langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle$, represents a volatility value w, an incarnation value i, an initial behavior e_i, a quiescent/stable behavior e_q, a creation dependence map δ_0, and a behavior

$$
\begin{aligned}
(\delta_0 \oplus \delta_1)(t) \quad &= \begin{cases}
\delta_0(t) & \text{if } t \notin \mathrm{Dom}(\delta_1) \vee \\
 & \quad (\delta_0(t) = \langle\, w_0, i_0 \,\rangle \,\wedge\, \delta_1(t) = \langle\, w_1, i_1 \,\rangle \,\wedge\, i_0 > i_1) \\
\delta_1(t) & \text{if } t \notin \mathrm{Dom}(\delta_0) \vee \\
 & \quad (\delta_0(t) = \langle\, w_0, i_0 \,\rangle \,\wedge\, \delta_1(t) = \langle\, w_1, i_1 \,\rangle \,\wedge\, i_0 < i_1) \\
\langle\, \max(w_0, w_1), i \,\rangle & \text{if } \delta_0(t) = \langle\, w_0, i \,\rangle \,\wedge\, \delta_1(t) = \langle\, w_1, i \,\rangle \,\wedge\, \max(w_0, w_1) < \mathcal{S} \\
\text{undefined} & \text{otherwise}
\end{cases} \\
\mathit{consis}(\delta_0, \delta_1) \; &\Leftrightarrow \; \forall t : (\delta_0(t) = \langle\, w_0, i_0 \,\rangle \,\wedge\, \delta_1(t) = \langle\, w_1, i_1 \,\rangle) \,\Rightarrow\, i_0 = i_1 \\
\mathit{inval}(\delta_0, \delta_1) \; &\Leftrightarrow \; \exists t : (\delta_0(t) = \langle\, w_0, i_0 \,\rangle) \,\wedge\, (\delta_1(t) = \langle\, w_1, i_1 \,\rangle) \,\wedge\, (i_0 > i_1) \,\wedge\, (w_1 < \mathcal{S}) \\
\mathit{stable}(\delta) \; &\Leftrightarrow \; \forall t : \delta(t) = \langle\, w, i \,\rangle \,\Rightarrow\, w > \mathcal{V}
\end{aligned}
$$

Fig. 2. Definitions of dependence map operations.

dependence map δ_1.

3.2.2 Dependence Maps

Dependence maps carry information regarding a transactor's creation and subsequent behavior changes induced by message reception. An *empty* dependence map, $\emptyset$, represents no dependencies on external transactors. Non-empty dependence maps carried along with messages enable transactors to determine, in a *lazy* manner, when partner transactors have rolled-back, potentially causing a local rollback behavior; when partners have become quiescent, potentially enabling local stabilization; when partners have become stable, effectively eliminating dependencies on such partner; when messages are invalid, due to previously received messages with a larger partner incarnation value; or when the sender transactor is a previously unknown partner effectively creating a new behavior dependence.

In order to facilitate reasoning about dependencies, we require the following operations on dependence maps:

$\delta_0 \oplus \delta_1$: union of dependence maps

$\mathit{consis}(\delta_0, \delta_1)$: tests if δ_0 is consistent with δ_1, i.e., it represents a valid incoming message

$\mathit{inval}(\delta_0, \delta_1)$: tests if δ_0 invalidates δ_1, i.e., it implies a rollback must happen in the receiving transactor.

$\mathit{stable}(\delta)$: tests if δ enables transactor stabilization, i.e., there are no pending dependencies on other transactors.

Definition **(** $\oplus$, $\mathit{consis}(\cdot,\cdot), \mathit{inval}(\cdot,\cdot), \mathit{stable}(\cdot)$**):** Given dependence maps $\delta, \delta_0, \delta_1 \in \mathsf{D}$, we define the dependence map union, $\delta_0 \oplus \delta_1$, the consistency test, $\mathit{consis}(\delta_0, \delta_1)$, the invalidating test, $\mathit{inval}(\delta_0, \delta_1)$, and the stability test, $\mathit{stable}(\delta)$ as depicted in Fig. 2.

Dependence map union is an associative and commutative operator with $\emptyset$ as identity. It represents the new dependence map resulting from combining its two operands in such a way that the most up-to-date dependence information is kept; in particular, notice that if a transactor is known to be stable by either of the original dependence maps, the union does not define a mapping for such stable transactor since by definition a stable transactor introduces no new dependencies. The consistency test determines whether two dependence maps are consistent; inconsistencies could arise from communications with either older or newer incarnations of a transactor, representing either invalid, out-of-order messages or transactor rollbacks respectively. The invalidating test determines whether the first dependence map renders the second invalid. A non-stable transactor is invalidated when it receives a message from a partner with a new incarnation value. Finally, the stability test succeeds when all dependencies in a map represent non-volatile transactors.

3.2.3 Transactor Configurations

A transactor configuration models a transactor system with a transactor state map, messages in transit, and a transactor stability information map. We define transactor configurations as follows.

Definition (Transactor Configurations): A *transactor configuration* with transactor state map, τ, multi-set of messages, μ, and stability information map, σ, is written $\left\langle \tau \mid \mu \mid \sigma \right\rangle$, where $\tau \in \mathsf{X} \xrightarrow{\mathrm{f}} \mathsf{E}$, $\mu \in \mathbf{M}_\omega[\mathsf{M}]$, $\sigma \in \mathsf{X} \xrightarrow{\mathrm{f}} \mathsf{S}$, and $\mathrm{Dom}(\sigma) = \mathrm{Dom}(\tau)$.

3.2.4 Single-step Transition Relation

There are three kinds of transitions between transactor configurations:

(i) **Local transitions** model transactor behavior as in sequential functional programs

(ii) **Transactor transitions** model transactor primitive operations - transactor creation, message sending and reception, stabilization, rollback and quiescence.

(iii) **Failure transitions** model failures in the computing environment.

The **local transition `fun`** is inherited from the purely functional fragment of our transactor language. The transition represents progress inside a single transactor.

The **transactor transitions** are:

`new`: creation of a transactor, returning its name.

`send`: message send, passes the message to the mail delivery system.

`receive`: message reception by a transactor.

`quiesce`: enters a *quiescent* state—it becomes ready with self-immutable, and persistent behavior. It may still rollback due to dependencies from other transactors.

`stabilize`: attempts to become ready with immutable, persistent, and consistent behavior.

`rollback`: rolls the transactor back to its initial behavior.

`volatility`: returns the transactor's volatility state

The **failure transitions** represent unreliable nodes and networks:

`lose`: loss of a message due to unreliable communication.

`reset`: recreation of a transactor state from persistent storage after a hardware reboot.

To describe the transactor transitions between configurations other than message receipt, a non-value expression is decomposed into a reduction context filled with a redex. Reduction contexts are expressions with a unique hole, and serve the purpose of identifying the subexpression of an expression that is to be evaluated next. Reduction contexts correspond to the standard reduction strategy (left-first, call-by-value) of Plotkin [20] and were first introduced by Felleisen and Friedman [10]. We use the symbol '$\square$' to denote the hole occurring in a reduction context, and call such holes *redex holes*.

Definition ($\mathsf{E}_{\mathrm{rdx}}, \mathsf{R}$): The set of *redexes*, $\mathsf{E}_{\mathrm{rdx}}$, and the set of *reduction contexts*, R, are defined by:

$$\mathsf{E}_{\mathrm{rdx}} = \mathsf{app}(\mathsf{V}, \mathsf{V}) \cup \mathsf{F}_n(\mathsf{V}^n)$$

$$\mathsf{R} = \{\square\} \cup \mathsf{app}(\mathsf{R}, \mathsf{E}) \cup \mathsf{app}(\mathsf{V}, \mathsf{R}) \cup \mathsf{F}_{n+m+1}(\mathsf{V}^n, \mathsf{R}, \mathsf{E}^m)$$

We let R range over R and r range over $\mathsf{E}_{\mathrm{rdx}}$.

An expression e is either a value or it can be decomposed uniquely into a reduction context filled with a redex. Thus, local transactor computation is deterministic.

Lemma (Unique decomposition): Either $e \in \mathsf{V}$, or $(\exists! R, r)(e = R[r])$.

Proof : An easy induction on the structure of e. $\square$

The purely functional redexes inherit the operational semantics from the purely functional fragment of our transactor language. The transactor redexes are: $\texttt{new}(e)$, $\texttt{send}(t, v)$, $\texttt{ready}(v)$, $\texttt{quiesce}(e)$, $\texttt{stabilize}()$, $\texttt{rollback}()$, and $\texttt{volatility}()$.

Definition ($\mapsto$): Figures 3 and 4 depict the single-step transition relation $\mapsto$ on transactor configurations.

The rules depicted in Fig. 3 describe the behavior of transactors in an idealized world where both networks and processors are perfectly reliable. These transition rules reflect a transactor model with support for global consistent states by tracking dependencies induced by message passing. Notice that the semantics does not enforce any particular locking or stabilization algorithm. It is up to higher-level application layers to provide efficient locking and stabilization protocols. The semantics does enforce, however, that once a transactor becomes stable, its state is consistent with other transactors' states, and it

`<fun>`

$$e \overset{\lambda}{\mapsto} e' \;\Rightarrow\; \left\langle \tau\{t \to e\} \mid \mu \mid \sigma \right\rangle \mapsto \left\langle \tau\{t \to e'\} \mid \mu \mid \sigma \right\rangle$$

`<new>`

$$\left\langle \tau\{t \to R[\mathtt{new}(e)]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\left\langle \tau\{t \to R[t'], t' \to e\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle, t' \to \langle\, \mathcal{V}, 0, e, \mathtt{nil}, \{t \mapsto \langle\, w, i \,\rangle\}, \emptyset \rangle\} \right\rangle \qquad t' \text{ fresh}$$

`<send>`

$$\left\langle \tau\{t \to R[\mathtt{send}(v_0, v_1)]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\left\langle \tau\{t \to R[\mathtt{nil}]\} \mid \mu, \mathtt{<} v_0 \Leftarrow v_1 \mathtt{>}_{\delta_0 \oplus \delta_1\{t \mapsto \langle\, w, i \,\rangle\}} \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle$$

`<receive>`

$$\left\langle \tau\{t \to \mathtt{ready}(v)\} \mid \mathtt{<} t \Leftarrow v_1 \mathtt{>}_\delta, \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\begin{cases} \left\langle \tau\{t \to \mathtt{app}(v, v_1)\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{V}, i, e_i, e_q, \delta_0, \delta_1 \oplus \delta \,\rangle\} \right\rangle & \text{if } w = \mathcal{V}, \text{ and } \mathit{consis}(\delta, (\delta_0 \oplus \delta_1\{t \mapsto \langle\, w, i \,\rangle\})) \\ \left\langle \tau\{t \to \mathtt{app}(e_q, v_1)\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{Q}, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{if } w = \mathcal{Q}, \text{ and } \mathit{consis}(\delta, (\delta_0 \oplus \delta_1\{t \mapsto \langle\, w, i \,\rangle\})) \\ \left\langle \tau\{t \to \mathtt{app}(e_q, v_1)\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{S}, i, e_i, e_q, \emptyset, \delta \,\rangle\} \right\rangle & \text{if } w = \mathcal{S}, \text{ and } \mathit{consis}(\delta, (\delta_0 \oplus \delta_1\{t \mapsto \langle\, w, i \,\rangle\})) \\ \left\langle \tau\{t \to \mathtt{app}(e_i, v_1)\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{V}, i + \mathit{1}, e_i, e_q, \delta_0, \delta \,\rangle\} \right\rangle & \text{if } w \neq \mathcal{S}, \text{ and } \mathit{inval}(\delta, \delta_1) \text{ (rollback)} \\ \left\langle \tau \mid \mu \mid \sigma \right\rangle & \text{if } w \neq \mathcal{S}, \text{ and } \mathit{inval}(\delta, \delta_0) \text{ (reset)} \\ \left\langle \tau\{t \to \mathtt{ready}(v)\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{otherwise (ignore)} \end{cases}$$

`<quiesce>`

$$\left\langle \tau\{t \to R[\mathtt{quiesce}(e)]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\begin{cases} \left\langle \tau\{t \to R[\mathtt{true}]\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{Q}, i, e_i, e, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{if } w = \mathcal{V} \\ \left\langle \tau\{t \to R[\mathtt{true}]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{otherwise} \end{cases}$$

`<stabilize>`

$$\left\langle \tau\{t \to R[\mathtt{stabilize}()]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\begin{cases} \left\langle \tau\{t \to R[\mathtt{true}]\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{S}, i, \mathtt{nil}, e_q, \emptyset, \emptyset \,\rangle\} \right\rangle & \text{if } w \neq \mathcal{V}, \text{ and } \mathit{stable}(\delta_0 \oplus \delta_1) \\ \left\langle \tau\{t \to R[\mathtt{nil}]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{otherwise} \end{cases}$$

`<rollback>`

$$\left\langle \tau\{t \to R[\mathtt{rollback}()]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\begin{cases} \left\langle \tau\{t \to e_i\} \mid \mu \mid \sigma\{t \to \langle\, \mathcal{V}, i + \mathit{1}, e_i, e_q, \delta_0, \emptyset \,\rangle\} \right\rangle & \text{if } w = \mathcal{V} \\ \left\langle \tau\{t \to R[\mathtt{nil}]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{otherwise} \end{cases}$$

`<volatility>`

$$\left\langle \tau\{t \to R[\mathtt{volatility}()]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\left\langle \tau\{t \to R[w]\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle$$

Fig. 3. The single-step transition relation $\mapsto$ on transactor configurations (assuming perfectly reliable networks and processors).

<lose>

$$\left\langle \tau \mid \texttt{<} t \Leftarrow v_1 \texttt{>}_\delta, \mu \mid \sigma \right\rangle \mapsto \left\langle \tau \mid \mu \mid \sigma \right\rangle$$

<reset>

$$\left\langle \tau\{t \to e\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle \mapsto$$

$$\begin{cases} \left\langle \tau \mid \mu \mid \sigma \right\rangle & \text{if } w = \mathcal{V} \\ \left\langle \tau\{t \to e_q\} \mid \mu \mid \sigma\{t \to \langle\, w, i, e_i, e_q, \delta_0, \delta_1 \,\rangle\} \right\rangle & \text{otherwise} \end{cases}$$

Fig. 4. Additional rules of $\mapsto$ modeling potentially unreliable networks and processors.

becomes immutable and persistent. The semantics also guarantees that once a transactor becomes quiescent, its state becomes self-immutable and persistent. Furthermore, to quiesce is a local decision and the transactor's state can only be rolled back by other transactors' invalidating messages.

The rules depicted in Fig. 4 model an unreliable network—a network where messages may get lost, or actors may fail due to computer reboots and crashes. The `<lose>` transition represents the loss of a message in the message delivery system. The `<reset>` transition represents the loss of a transactor due to hardware failures. Notice that stable and quiescent transactors recover their state from persistent storage, while volatile transactors completely disappear.

4 Distributed Transaction Example

Figures 5 and 6 describe a somewhat more realistic and complete transactor example than that given in Figure 1. In this example, there are two types of transactors: A `BuySell` transactor, depicted in Fig. 5 represents an agent that can either buy or sell a commodity. Typically, two or more `BuySell` transactors will interact with one another to complete a sale. The `Broker` transactor depicted in Fig. 6 serves to bring two `BuySell` participants together. Note that the existence of a "middleman" is not essential; we could have designed a similar application with direct communication between buyer and seller, at the cost of some clarity in the specification.

The idea of using a chain of transactors to model sequences of committed states as described in the `Counter` example of Section 2.1 is used again here for the committed states of `BuySell` participants.

A sales transaction is initiated with a participant using the `initiate(...)` message processor. As in the `Counter` example, `initiate(...)` chains through a sequence of committed transactors until a volatile transactor is reached. At this point, the code checks whether the requested inventory and price adjustments are feasible. If not, the participant informs the broker that the sale cannot complete, then rolls back. If the transaction is feasible, the participant updates its state appropriately, creates a new transactor to handle subsequent updates, and then executes the `quiesce` primitive. Once a transactor is quiescent, it may neither update its state nor roll back (the transactor operational semantics treats assignments or the `rollback` primitive as no-ops when the transactor is stable or quiescent). Also, should a quiescent transactor fail, if it ever recovers, it is guaranteed to recover with the state it had prior to failure. Although a quiescent transactor t_1 cannot change its own state, if it receives a message from another transactor t_2 that is inconsistent with messages received from t_2 prior to quiescing (because t_2 has rolled back), t_1 will roll back. The quiescent state is thus similar to the "prepared" state of a 2-phase commit protocol: it indicates that the quiescent transactor is prepared to commit its current state in a recoverable way, but is also able to roll back if its partner transactors are unable to complete the transaction.

The `Broker` transactor in Fig. 6 serves to bring two participants together, and checks whether both are capable of completing the transaction before allowing the participants to stabilize. To make the example slightly more realistic, the `Broker` also spawns off an auxiliary `Timer` transactor, whose sole purpose is to call the `Broker` back after a predetermined length of time has elapsed. If the two `BuySell` participant transactors do not communicate back with the `Broker` before `Timer` sends the `Broker` a `timeout()` message, the `Broker` will roll back and abort any active participants.

After a `BuySell` transactor is quiescent, it waits for a `complete_sale()` or an `abort()` message from the `Broker`. In the former case, the `Broker` indicates that all participants are capable of completing the transaction. After receiving the `complete_sale()` message, the `BuySell` participant then executes the `stabilize` primitive and sends a `ping()` message to its volatile child. The dependence information piggybacked on this otherwise vacuous message informs the child transactor that the parent is stable, which is a prerequisite for the child's stabilization.

When a `BuySell` transactor receives an `abort()` message, it first sends a `ping_request()` message to the `Broker`. The `Broker` then replies with an empty `ping()` message, which serves to communicate the `Broker`'s dependence information. Since the `Broker` rolls back immediately after sending `abort()` messages to the `BuySell` participants, the dependence information associated with the `ping()` will force the participant itself to roll back, as a result of its inconsistency with the (rolled-back) state of the `Broker`. The use of the `ping_request()` and `ping()` messages by the `abort()` message processor ensures that the participant will roll back even if it is quiescent (and hence unable to initiate rollback on its own).

There are a few other aspects of `BuySell` and `Broker` that are worth noting:

- There is nothing to prevent multiple `Broker` transactors from sending messages to the same `BuySell` participant. Thus, e.g., a `Broker` b_1 could initiate a transaction, while a second `Broker` b_2 could call, e.g., `abort()` or complete_sale(). We could explicitly prevent this in various ways, e.g., by ensuring that the identity of a participant is only made available to one broker at a time, or recording the identity of the Broker that initiates a transaction, recording the Broker's identity in the participant's state, then adding an extra Broker argument to the other message processors as an authentication mechanism to ensure that the only one Broker can participate in a transaction at a time. However, the transactor operational semantics will automatically prevent certain forms of misuse; for example, if a Broker b_2 sends the abort() message to a participant involved in a transaction initiated by Broker b_1, the result will be a no-op since abort() only rolls back the participant's state if called from a transactor in a state inconsistent with the dependence information captured by a previous interaction with the participant.

- As written, Broker is somewhat resilient to a failure to receive messages from participants, due to the timeout mechanism. However, if a BuySell participant fails to receive a complete_sale() message from a broker, it could remain in a quiescent, but not stable state indefinitely. A timeout or message retry mechanism could be added to BuySell to make it more resilient to failures.
- In general, transactors provide no predefined protocols for stabilization, synchronization, or recovery from message or node failure. Instead, the primitives ensure that interacting transactors never reach mutually inconsistent states, and provide sufficient information to allow a variety of protocols to achieve consistent (stable) states when needed (but not until needed).

5 Discussion and Future Work

While the transactor semantics of Section 3 is useful for defining key transactor concepts, it has several shortcomings that we wish to address:

- The structure of dependence information is too *coarse* for many applications. For example, when sending a message consisting of a pair of values, we could in principle decompose the message's dependence information into a corresponding pair. If the receiving transactor's message processor only reads one element of the pair, no dependences need to be induced on the unread element. The state of a transactor could also be broken down into finer-grained elements, with dependences for each element tracked separately.
- While we found it convenient for design purposes to base our semantics on the actor semantics of Agha et al. [3], this semantics does not clearly distinguish the immutable "program" controlling a particular transactor from the "state" of the actor, which can evolve as each message is processed. Both of these logically distinct concepts are encoded in the same lambda expression. By adopting a semantics that makes the distinction between these concepts

```
// BuySell:
//
// Participant transactor in a sales transaction.
// Assumes that only one broker instance will send
// the participant messages while a transaction is
// being negotiated, and that all messages sent will
// be received.
//
BuySell(int init_inventory, int init_cash_balance) {

  // current inventory and cash on hand
  int inventory    = init_inventory;
  int cash_balance = init_cash_balance;

  // next_val is non-Null if this value is committed
  // (stable); end of chain of transactors rooted at
  // next_val yields last committed value
  BuySell next_val = Null;

  // initiate(broker, inv_adj, cash_adj)
  //
  // initiate a sales transaction brokered by broker,
  // which requests that the participant's inventory be
  // incremented by inv_adj, and that its cash balance
  // be incremented by cash_adj
  //
  initiate(Broker broker, int inv_adj, int cash_adj) {
    if ( volatility == stable )
    // this is a committed (stable) value; find first
    // uncommitted value in chain
      next_val.initiate(broker, inv_adj, cash_adj);
    } else if (inventory + inventory_adj < 0 ||
      cash_balance + cash_adj < 0) {
    // inventory and/or cash_balance inadequate to
    // complete transaction
      broker.no_sale(this);
      rollback;
    }
    else {
      inventory += inventory_adj;
      cash_balance += cash_adj;
      next_val = new BuySell(inventory, cash_balance);
      quiesce;
      broker.ready(this);
    }
  }

  // complete_sale()
  //
  // broker uses this message to indicate that
  // all parties have agreed to complete the sale
  //
  complete_sale() {
    // the stabilization attempt should always succeed
    // when BuySell is used with Broker
      stabilize;
    // ping child value to communicate stabilized
    // status
      next_val.ping();
  }

  // abort()
  //
  // confirms abort request by sending ping request to
  // Broker, which should respond with a ping (see
  // below)
  //
  abort() {
    broker.ping_request(this);
  }

  // ping()
  //
  // trivial message processor used solely to receive
  // (implicit) status information from another
  // transactor (here, either another BuySell
  // transactor or a Broker transactor); such
  // information may cause the transactor to roll
  // back (e.g., if Broker has rolled back)
  //
  ping() {}
}
```

Fig. 5. Participant in a sale transaction.

clearer, we can, e.g., distinguish "stateless" transactors, whose state does not change with each message processed, from stateful ones. This distinction can in turn be used to eliminate certain spurious dependences.

- In general, our model may require that dependence sets of unbounded size be maintained in a transactor's volatility state. We conjecture that type systems or similar annotations could be used to ensure that only bounded dependence sets need be maintained in many realistic cases.

A number of questions remain open regarding the proposed transactor model:

- Should a transactor be able to explicitly inspect its dependence information?
- Should a transactor (as opposed to a transactor reference) be a "first-class" value?
- Does a kernel coordination language require explicit support for authentication (note that the possession of a transactor reference constitutes a sort of "capability")?
- Should selective disablement of message processors be supported [13]? (e.g., as a locking mechanism for *sequences* of operations)

```
// Broker:
//
// Brings buyer and seller together to perform a sale
// transaction.  Assumes that all messages sent will be
// received.
//
Broker(BuySell buyer, BuySell seller,
       int num_items, int sale_price)

  // flags set to true when participant has
  // successfully completed its part of transaction
  bool buyer_ready   = false;
  bool seller_ready  = false;

  // do_sale()
  //
  // initiate sale transaction
  //
  do_sale() {
    buyer.initiate(this, num_items, -sale_price);
    seller.initiate(this, -num_items, sale_price);
    // timer ensures that broker doesn't wait
    // indefinitely for participants to complete their
    // part of the transaction
    (new Timer()).callBackIn(10, this);
  }

  // ready(partner)
  //
  // sent by partner to indicate that it is committed to
  // completing its part of the transaction
  //
  ready(BuySell partner) {
    if ( partner == buyer ) buyer_ready = true;
    if ( partner == seller ) seller_ready = true;
    if ( buyer_ready && seller_ready ) {
      // stabilization should succeed at this point,
      // because participants are quiescent
      stabilize;
      buyer.complete_sale();
      seller.complete_sale();
      system.print("sale successful");
    }
  }

  // no_sale(partner)
  //
  // sent by partner to indicate that it is unable
  // complete its part of the transaction
  //
  no_sale(BuySell partner) {
    if ( partner = buyer ) {
      seller.abort();
      system.print("buyer aborted sale");
      rollback;
    }
    if ( partner = seller ) {
      buyer.abort();
      system.print("seller aborted sale");
      rollback;
    }
  }

  // ping_request()
  //
  // implicitly sends Broker's status (i.e., the
  // Broker's dependence map information) to requester
  // using an empty message
  //
  ping_request(BuySell requester) {
    requester.ping()
  }

  // timeout()
  //
  // called by auxiliary timer transactor when time to
  // complete transaction has elapsed
  //
  timeout() {
    if ( ! (buyer_ready && seller_ready) ) {
    // abort sale if participants haven't responded
      buyer.abort();
      seller.abort();
      system.print("timeout before sale complete");
      rollback;
    }
  }
}
```

Fig. 6. Broker for a sale transaction.

- What is the right set of high-level "reliable" programming abstractions to build on top of transactors?

Acknowledgments

The authors would like to thank James Leifer for detailed comments on previous drafts of this paper, and the anonymous FOCLASA referees for helpful feedback.

References

[1] Agha, G., N. Jamali and C. Varela, *Agent Naming and Coordination: Actor Based Models and Infrastructures*, in: A. Ominici, F. Zambonelli, M. Klusch and R. Tolksdorf, editors, *Coordination of Internet Agents*, Springer-Verlag, 2001 pp. 225–248.

[2] Agha, G. and N. Jamali, *Concurrent programming for distributed artificial intelligence*, in: G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, MIT Press, 1999 .

[3] Agha, G., I. A. Mason, S. F. Smith and C. L. Talcott, *A foundation for actor computation*, Journal of Functional Programming **7** (1997), pp. 1–72.

[4] Agha, G., "Actors: A Model of Concurrent Computation in Distributed Systems," MIT Press, 1986.

[5] Bernstein, P. A., *Middleware: A model for distributed system services*, Communications of the ACM **39** (1996), pp. 86–98.

[6] Birman, K. and R. Renesse, *Reliable distributed computing with the isis toolkit* (1994).

[7] Cardelli, L. and A. Gordon, *Mobile ambients*, in: *Foundations of System Specification and Computational Structures*, LNCS 1378, Springer Verlag, 1998 pp. 140–155.

[8] Ciancarini, P. and C. Hankin, editors, "First International Conference on Coordination Languages and Models (COORDINATION '96)," Springer-Verlag, Berlin, 1996.

[9] Ciancarini, P. and A. Wolf, editors, "Third International Conference on Coordination Languages and Models (COORDINATION '99)," Springer-Verlag, Berlin, 1999.

[10] Felleisen, M. and D. Friedman, *Control operators, the secd-machine, and the λ-calculus*, in: M. Wirsing, editor, *Formal Description of Programming Concepts III*, North-Holland, 1986 pp. 193–217.

[11] Fournet, C. and G. Gonthier, *The reflexive cham and the join-calculus* (1996).

[12] Frølund, S. and G. Agha, *A language framework for multi-object coordination*, in: *Proceedings of ECOOP 1993*, Springer Verlag, 1993 LNCS 707.

[13] Frølund, S., "Coordinating Distributed Objects: An Actor-Based Approach to Synchronization," MIT Press, 1996.

[14] Garlan, D. and D. L. Metayer, editors, "Second International Conference on Coordination Languages and Models (COORDINATION '97)," Springer-Verlag, Berlin, 1997.

[15] Hewitt, C., *Viewing control structures as patterns of passing messages*, Journal of Artificial Intelligence **8-3** (1977), pp. 323–364.

[16] Kim, W. and G. Agha, *Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages*, in: *Proceedings of Supercomputing'95*, 1995.

[17] Liskov, B., *Distributed programming in argus*, Communications of the Association of Computing Machinery **31** (1988), pp. 300–312.

[18] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, parts I-II*, Information and Computation **100** (1992), pp. 1–77.

[19] Object Management Group, *CORBA services: Common object services specification version 2.*, Technical report, Object Management Group (1997), http://www.omg.org/corba/.

[20] Plotkin, G., *Call-by-name, call-by-value and the lambda calculus*, Theoretical Computer Science **1** (1975), pp. 125–159.

[21] Specification, X. C., *Distributed transaction proceesing: The XA specification*, X/Open Company Limited, xO/CAE/91/300.

[22] Spector, A., R. Pausch and G. Bruell, *Camelot: A flexible, distributed transaction processing system*, in: *Proc. IEEE Computer Society International Conf.* (1988), pp. 432–437.

[23] Sun Microsystems Inc. – JavaSoft, *Remote Method Invocation Specification* (1996), work in progress. http://www.javasoft.com/products/jdk/rmi/.

[24] Talcott, C. L., *Composable semantic models for actor theories*, Higher-Order and Symbolic Computation **11** (1998).

[25] van Renesse, R., K. P. Birman and S. Maffeis, *Horus: A flexible group communication system*, Communications of the ACM **39** (1996), pp. 76–83.

[26] Varela, C. and G. Agha, *A Hierarchical Model for Coordination of Concurrent Activities*, in: P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594 (1999), pp. 166–182, http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[27] Waldo, J., *JINI Architecture Overview* (1998), work in progress. http://www.javasoft.com/products/jini/.