Programming Live Distributed Objects with Distributed Data Flows

Krzysztof Ostrowski Ken Birman
Cornell University, Ithaca, NY 14853, USA
{krzys, ken}@cs.cornell.edu

Danny Dolev
Hebrew University, Jerusalem 91904, Israel
dolev@cs.huji.ac.il

Abstract

This paper presents a new object-oriented approach to modeling the semantics of distributed multi-party protocols such as leader election, distributed locking, or reliable multicast, and a programming language that supports it. The approach builds on and extends our *live distributed objects* model [37] by introducing a new concept of a *distributed flow*, a stream of messages flowing concurrently at multiple locations. Our flows correspond to variables, private fields, and method parameters in Java-like languages; they are the means by which we store and communicate state. Active protocol instances, which correspond to Java objects, consume and output flows; their internal states are encapsulated as internal flows, and all of their internal logic is represented as operations on flows.

Our language supports a new type of concern separation: the semantic structure of protocols is decoupled from implementation details such as the construction and maintenance of overlays, trees, or other hierarchical structures needed for scalability. The latter can be addressed by the compiler or at the deployment time; it can be done differently in different parts of the network, to match local network characteristics.

The paper introduces the basic language concepts, syntax, and semantics, illustrating formal definitions with a discussion of example protocols such as leader election, distributed locking, agreement, and loss recovery. It shows examples of rules for a formal reasoning about programs in our language.

While full implementation details of the supporting compiler and runtime are beyond the scope of this paper, we do briefly describe how our new language primitives can be implemented. Our approach is practical: the core language constructs, including hierarchical monotonic aggregations, have been implemented and evaluated in a simulator [38]. The full compiler framework is in preparation and will be publicly released as a part of our *live distributed objects* platform [1].

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed Programming; D.1.5 [*Programming Techniques*]: Object-Oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Reliability, Theory

Keywords Distributed Data Flows, Distributed Multi-Party Protocols, Live Distributed Objects, Monotonic Aggregation

1. Introduction

The premise of this work is that *distributed multi-party protocols* (DMP) such as virtual synchrony [7], two-phase commit [43], or Paxos [30] are becoming increasingly important and used pervasively, and that further advances will require that developers be able to design their own DMPs. Our goal is to provide a simple, yet expressive protocol definition language that allows developers to express desired DMP semantics concisely, using high-level constructs. We'd like the logical flow of protocol state and decisions to be readily understood from the code, not obfuscated by low-level operations such as sending individual network messages from A to B.

Programming DMPs is inherently difficult [40], but it can be simplified by tools that promote a separation of concerns. Developers should be able to specify the semantics and *logical* control flow without having to explicitly handle *physical* aspects, such as failures, timeouts, network topology, and organizing nodes into trees, rings, or other scalable structures; the latter can and should be treated as orthogonal, much as compiler optimizations in C++ are orthogonal to the semantics of code. To enable this, we need a set of programming abstractions that are powerful enough to express commonly used DMPs, but that leave enough flexibility for the compiler to generate scalable code. This inherent tension between expressiveness and compiler flexibility has been the key factor that shaped our approach and our design decisions, and that distinguishes this work from the existing protocol languages.

Before going further, let's elaborate on some of the points we made earlier. First, we've stated that DMPs are becoming increasingly important, and used pervasively. In the past few decades, DMPs have been used mostly in data centers, financial institutions, or military settings, for example, to replicate services and data, for load-balancing or fault-tolerance [33], or to coordinate configuration changes and synchronize access to services [9]. In this model, DMPs ran mostly among servers in data centers, whereas the larger Web has remained predominantly client-server: home user's machines wouldn't communicate with each other. In other work [36], we argued that this is bound to change. Home user's computers, equipped with ever-increasing amounts of memory and multi-core CPUs, are getting faster, whereas web content providers are stumbling over scalability as their users bases expand. Many classes of dynamic, interactive, short-lived content (collabo-

rative work, interactions in virtual worlds and online games) can't easily be cached and indexed, and may be hard to scale by adding more servers. It is only natural to off-load servers by pushing data out of data centers, and towards the clients.

Technologies based on this idea already exist. In our *live distributed objects* (LO) platform [1], every visual element on an interactive web page – a chat window, a video stream, a shared document – can be individually powered by a DMP; its content doesn't reside on a remote server; it is replicated among the clients, in a peer-to-peer fashion. The DMP running among the clients ensures that all replicas stay in sync[1]. The creators of Smalltalk [24] used similar approach as a basis of their Croquet [44] platform; 3D objects in their virtual space are replicated with a variant of 2PC [43]. Darkstar [45] and several other [13, 46] projects also fall into this category. Each of these technologies leads to a pervasive use of DMPs.

The second premise of our work is that programmers will want to build their own DMPs. Distributed computing forces them to choose between reliability, scalability, performance and persistence, and different applications require a different balance. For example, the version of reliable multicast DMP used for database replication in a financial institution would require a consensus semantics, but wouldn't need to scale to thousands of nodes, whereas the variant of reliable multicast used to synchronize players in an online multi-player game (MMORPG), or clients watching a streaming movie, would require excellent scalability at the cost of weaker guarantees. In other work [36], we pointed out that even for a seemingly simple task such as collaborative editing, there exists a surprising variety of different approaches that rely on different ways of locking, reconciliation, or flavors of multicast, often fine-tuned to the particular application domain. The analogy to Java or .NET collections seems appropriate: even though many applications do not require custom collections, and can be built using the small set of standard abstractions, such as lists, arrays, or hash tables, those who build high-performance or scalable systems often design their own custom collections optimized for their specific applications. Compared to collections, DMPs and their tradeoffs can be even more complex and diverse. Hence, this type of flexibility is essential.

Designing DMPs in languages such as Java is hard; popular toolkits like Ensemble [20], Spread [3], and Appia [34] have 25,000+ lines of code. Systems such as MACE [26] can remove much of the common programming burden, but programmers still have to think at the level of states, transitions, and network messages sent between pairs of nodes; this may be easy for loosely-coupled systems such as distributed hash tables (DHT), but it can be hard for DMPs. One way to simplify the process is by composing pre-existing reusable protocol layers in DMP composition toolkits such as Ensemble, Spread, Appia, or BAST [17]. The latter approach is convenient, but it has its limitations. First, to achieve a high degree of flexibility, one needs a very large number of thin and simple protocol layers: Ensemble has 50; even then, flexibility is limited, for only certain combinations of layers make sense. Flexibility in these systems generally amounts to including (or not) certain functional layers, e.g., ordering, whereas to use a *different* ordering scheme, one generally has to develop a custom layer in Java; this, in turn, requires familiarity with the architecture of the DMP composition toolkit and its API. Finally, while the toolkits separate different functional layers from one-another, their functionality is often tightly coupled to implementation; for example, a layer that handles recovery, ordering, or stability may be hard-wired to aggregate its information in a particular manner, such as by using a leader or an *all-to-all* communication pattern, and may be unable to easily switch to gossip or structures such as trees and rings. The latter weakness applies also to MACE and other systems that require the programmer to work at the level of state transitions and network packets; code that maintains distributed structures becomes intermingled with and essentially inseparable from the core semantics and logical information flow.

In this paper, we advocate a radically different approach: we propose a few simple generic abstractions that can be easily composed to express semantics as diverse as distributed agreement and leader election, that can be stacked hierarchically to express scalable hierarchical protocols, and that can themselves be implemented in a variety of ways, such as by using token rings, trees, gossip, or IP multicast. Protocols in our language are compact and easy to reason about, while at the same time they leave the runtime a high degree of flexibility in mapping our language constructs to executable code.

Before continuing, it may be helpful to the reader to skim over examples of protocols to get the feel of our language. In the paper we present the code for distributed locking (Figure 6), loss recovery (Figure 10), leader election (Figure 17), barrier synchronization (Figure 18), distributed agreement (Figure 20), and atomic delivery (Figure 19), and hierarchical variants of leader election (Figure 23) and recovery (Figure 26). Note the use of set operators such as $\cup$ or **intersect** in the code of loss recovery, atomic delivery, and agreement, and the use of recursion in hierarchical examples. The use of aggregation to implement global decisions, set calculus for batched processing, and recursion for hierarchical scalability are the three core concepts that underpin our approach.

In order to fully explain our examples, we need to discuss the semantics of our programming constructs; in particular, the definitions and properties of flows (Section 2.2) and their dependencies (Section 2.4). For this reason, most examples are presented fairly late in the paper, starting on page 12. The reader may find it helpful to only skim over formal notation on the initial reading. We tried to illustrate all of the essential concepts through figures and concrete examples.

This paper makes the following contributions:

- It proposes a new programming abstraction, a *distributed data flow*, and describes the four basic types of operations that can be performed on flows: *disseminations*, *transfor-*

[1] We encourage the reader to watch the videos on our project's website [1].

mations, *aggregations*, and *distributions*. It discusses the formal properties and semantics of the new concepts, and explains their role using examples and illustrations.

- It proposes a new object-oriented programming language that operates on distributed flows. It describes its syntax and semantics and briefly explains how each of the mechanisms we are proposing can be physically implemented.
- It proposes a new approach to modeling strong semantics through monotonic aggregation: a new concept that is intuitive, very cheap to implement, and extremely versatile.
- It presents the code of a variety of protocols and explains how to reason about their semantics; in particular, it illustrates the practical role of different flavors of aggregation.
- It proposes a new use of the set arithmetic, as a means of expressing batched processing in distributed protocols. It briefly discusses a space-bounded variant of it used in our platform, and demonstrates its use in example protocols.
- It proposes a new use of recursion, as a means of expressing hierarchical architectures in distributed protocols. Using an example, it briefly explains how a simple recursive program in our language can be automatically expanded to form a complex hierarchical distributed structure.
- It describes a new type of concern separation enabled by our approach: the decoupling of the logical structure and semantics of distributed protocols from the construction and maintenance of hierarchical structures and the manner in which information is disseminated and aggregated.
- It shows how the global behavior of a distributed protocol can be modeled in a purely functional style: the four basic operations on flows could be viewed as purely functional.

2. Language

2.1 Objects

As noted earlier, the approach proposed here builds upon and extends our *live distributed objects* (LO) [37] model; hence, we start by introducing LO (for more detail, see [36, 37]).

Each physical machine participating in the execution of a DMP runs a piece of code (the protocol stack) that maintains some local state and interacts with local applications and the local OS. We refer to such running piece of code as a *proxy*. Each running instance of a DMP involves a group of proxies on multiple physical computers, sending network packets to one-another. We refer to this group of proxies collectively as a *live distributed object*, or a *live object* (LO) (Figure 1). LO is the basic unit of composition and means of encapsulation in our model; it serves similar purposes as an object in Java.

Method calls and callbacks between the applications and the DMP's protocol stack are modeled as *events* (*messages*). The API exposed by the DMP is modeled as a set of message channels. We refer to these as *endpoints*. The term endpoint *instance* refers to a particular message channel exposed by a single specific proxy. To say that object O exposes endpoint I means that every proxy of O exposes an instance of I, and

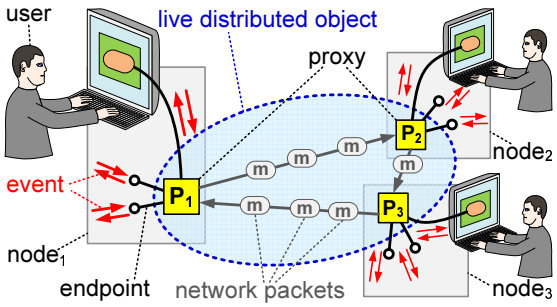

Figure 1. A *live distributed object* exists simultaneously in multiple locations: it consists of a group of communicating *proxies* (here P_1, P_2, and P_3), with their internal local states and all network packets flowing between them. The proxies' local states and network communication are the live object's internals, invisible from the outside. An object interacts with its software environment by passing *messages* via *endpoint instances*: local message channels exposed by all its proxies.

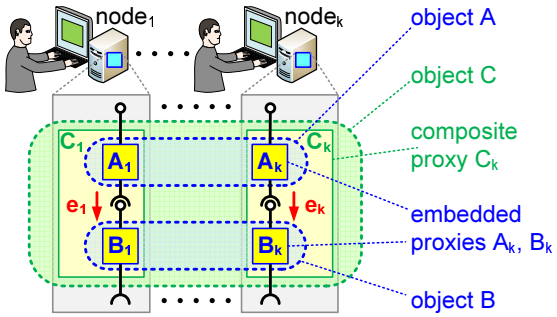

Figure 2. Live objects can be composed by connecting endpoint instances exposed by their proxies (here A, B are composed this way). A composite object (C) can have embedded objects (A, B). Proxies of these embedded objects (A_k, B_k) are encapsulated inside proxies of the composite object (C_k).

all instances carry messages of the same types. A proxy may expose multiple endpoint instances for a variety of purposes; in particular, to interact with proxies of application objects, proxies of objects that are recursively embedded (Figure 2), and infrastructure services. All interactions between a proxy and its environment are tunneled through endpoint instances.

LO can be recursively nested by embedding their proxies inside of one-another, and composed by connecting endpoint instances exposed by pairs of their proxies (Figure 2).

Referring to a running DMP instance as an *object* might at first seem awkward, but our objects are not much different from those in Smalltalk [24]; they interact via messages, and they may encapsulate internal state and threads of execution. The only difference is that *state* and *execution* encapsulated within a live object are *distributed* in the sense that they reside (occur) in multiple locations at a time: each proxy carries a portion of each logical unit of a live object's state, and each can take independent actions. This distributed perspective on state and execution is important; it underpins the distributed flow definition in Section 2.2, which is central to this paper.

Referring to instances of a DMP protocol stack as *proxies* stresses the fact that in our model, it's the distributed behavior that constitutes a service. Proxies thus aren't objects; they are gateways through which a local machine can gain access to (or participate) in a distributed behavior. In this sense, LO proxies generalize the concept of Java RMI proxy stub [49].

2.2 Flows

We define a *distributed data flow*, or simply a *data flow* (DF), as a set of messages of the given type appearing on instances of the given endpoint exposed by the given object (Figure 3). As mentioned earlier, the individual messages may represent method calls and callbacks between proxies of the object and their local environments. We assume that all messages in the given DF always flow in the same direction, i.e., either into or out of its object's proxies. To specify the direction, we call a DF an *input* or *output* flow of the given object, respectively. Note that one object's output flow will normally be an input flow of another object (Figure 3). Finally, if objects A, B are embedded in a composite object C (Figure 2), flows between A and B are said to be *encapsulated* in C (or *internal* to C).

Example 1. Suppose a certain application object A uses a distributed lock object B (Figure 3). Whenever a proxy of A running on node x wishes to acquire or release the distributed lock, it makes a call to its local proxy of B; we represent this as a message $wants(b)$ carrying a Boolean value $b \in \mathbb{B}$ ($\mathbb{B} = \{false, true\}$), $b = true$ if the lock is to be acquired, and $b = false$ if it is to be released. The set of all messages $wants(b)$ generated by A's proxies across all the nodes, and at any time, constitutes a distributed data flow from A to B. By convention, we name flows after the messages they carry, and the type of a flow is determined by the type of the values. In this case, a Boolean flow named $wants$ is an output of A, and an input to B; it carries the distributed information about the willingness to acquire the lock from proxies of A into B.

Similarly, suppose that each time a proxy of B invokes a callback to notify its local proxy of A that it was granted or denied the lock, we model this as a message $holds(b)$, where $b = true$ if the lock was granted, and $b = false$ otherwise. The set of all messages $holds(b)$, generated by B's proxies, again forms a Boolean flow, this time from B back to A. The flow $holds$ carries distributed information about the ownership of the lock. The locking object B can be thought of as a sort of distributed "transformation" of $wants$ into $holds$. ■

It might be useful to think of input and output flows as the analogues of formal parameters and return values of methods in Java; indeed, as the above example suggests, most objects generate output flows that carry various decisions, calculated in response to requests in the input flows. Likewise, internal flows may be thought of as the analogues of private fields of a Java class. This is true in a fairly literal sense: flows in our language are the means of storing and communicating state.

Each message $m \in \alpha$ in a flow α is formally modeled as a quadruple of the form $m = (x, t, k, v)$; x is the *location* at which the message flows, t is the *time* at which this happens,

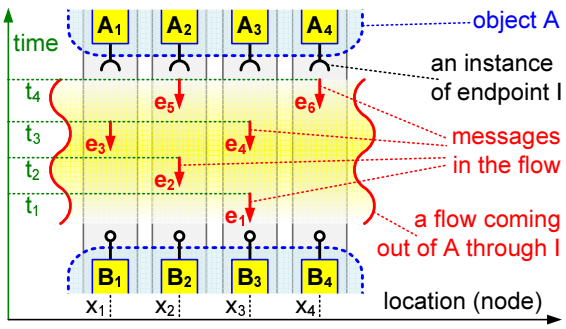

Figure 3. The flow leaving A through endpoint I consists of messages $\{e_1, e_2, \ldots, e_6\}$. Notice that the flow is distributed both in time (different t_i), and in space (different nodes x_j). Each message e_i may represent an asynchronous method call from some proxy of A to its (locally connected) proxy of B. Formally, each e_i is represented as a quadruple. For example, in this figure, $e_3 = (x_1, t_3, k, v)$ for some $k \in \mathcal{K}$ and $v \in \mathcal{V}$.

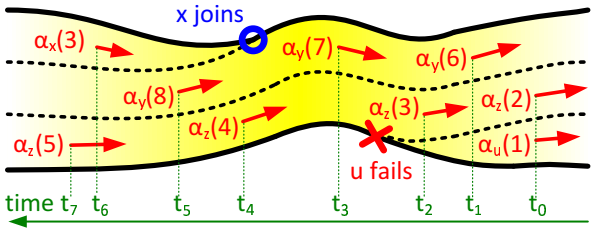

Figure 4. The set of locations in a distributed flow isn't static; the flow can begin and end at different locations as nodes join, leave, fail, and reboot. After the connection at location u is terminated, flow α stops at u, but it continues at other locations. After a new proxy starts and connects its endpoint instance, the flow expands to x. Each $\alpha_x(k)$ represents a value of some message $(x, t, k, v) \in \alpha$, for some $t \in \mathcal{T}, v \in \mathcal{V}$.

k is a *version* number it is tagged with, and v is the *value* it carries. Given message m, the elements of the quadruple are denoted as $\chi(m)$, $\tau(m)$, $\kappa(m)$, and $\nu(m)$, and the sets of all locations, times, versions, and values are denoted as $\mathcal{X}$, $\mathcal{T}$, $\mathcal{K}$, and $\mathcal{V}$, respectively. The set of all flows is denoted as $\mathcal{F}$; by definition, $\mathcal{F} \subset \mathcal{X} \times \mathcal{T} \times \mathcal{K} \times \mathcal{V}$. Not all sets of quadruples are considered flows; only those that occur in real systems[2]. Sets $\mathcal{T}$, $\mathcal{K}$, and $\mathcal{V}$ are assumed to be linearly[3] ordered by $\leq$[4].

For the sake of simplicity, we'll think of locations $x \in \mathcal{X}$ as physical nodes, although formally, each location identifies a single endpoint instance, and a single continuous period of time while this endpoint instance remains connected to some other endpoint instance. To understand this, consider Example 1. Every time the system is deployed onto a new node i,

[2] The complete formal model imposes a number of constraints that we omit to keep this presentation simple. For example, we assume that only finitely many messages can flow in a finite time interval and that $\mathcal{T}$ is well-founded.

[3] For the sake of simplicity, we'll assume linear ordering wherever possible. The definitions and theorems we use carry over to the partial ordering case.

[4] We treat $\leq$ and $<$ as "overloaded"; their meaning is clear from the context.

creates proxies A_i and B_i, and connects endpoint instances, this newly established connection represents a new location, some $x \in \mathcal{X}$. If proxy A_i sends a message $wants(b)$ to B_i to request the lock, this is modeled as $(x, t, k, b) \in wants$ for some $t \in \mathcal{T}, k \in \mathcal{K}$. If node i later crashes and reboots, or if for some other reason proxies A_i and B_i get disconnected from each other and later reconnected, their new connection will be considered a *different* location $x' \neq x$. This approach corresponds to the widely-adopted *fail-stop* model [41].

If the connection between some pair of endpoint instances is established and never broken (the node never crashes, the two proxies keep executing, and neither explicitly terminates the connection), the corresponding location $x \in \mathcal{X}$ is called *live* or *permanent*; otherwise, it is called *transient*. Locations at which messages appear in flow α are called the *locations of* α, denoted as $\mathcal{X}(\alpha)$, and formally defined as follows:

$$\mathcal{X}(\alpha) = \{x \in \mathcal{X} \mid \exists_{m \in \alpha}\, \chi(m) = x\} \,. \tag{1}$$

The set of all permanent locations of α is denoted as $\mathcal{X}^*(\alpha)$. For future use, we define $\mathcal{T}(\alpha)$, $\mathcal{K}(\alpha)$, $\mathcal{V}(\alpha)$ analogously, as the sets of all times, versions, and values of messages in α.

Since nodes can join, leave, or fail, and endpoint instances can be disconnected and reconnected, a flow can have an unbounded set of locations, but at any given point in time, messages continue to appear only at a finite number of locations (Figure 4). One might think of the flow as (locally) ending at some locations even as it expands onto the new ones.

We assume that $\mathcal{T}$ is a global, linearly ordered time. The time values $\tau(m)$ are not physically carried in messages, and are not observable; we use them only for modeling purposes. Only the version $\kappa(m)$ and value $\nu(m)$ are physically carried in a message m, and can be used as parts of the computation.

For now, it's best to think of versions simply as sequence numbers increasing on each message (the general case is discussed later). We assume that for the same location x, messages with higher versions flow later. Formally, for each message pair $m, m' \in \alpha$ flowing in $\alpha \in \mathcal{F}$, the following holds:

$$\chi(m) = \chi(m') \wedge \tau(m) < \tau(m') \Rightarrow \kappa(m) < \kappa(m') \,. \tag{2}$$

We also assume that at the same location x, messages tagged with the same version have the same value, as defined below:

$$\chi(m) = \chi(m') \wedge \kappa(m) = \kappa(m') \Rightarrow \nu(m) = \nu(m') \,. \tag{3}$$

The most important part of each message m is, of course, the value $\nu(m)$ stored in it, and in the remainder of the paper we often discuss how values in different messages are related to one-another. To simplify formulas, we use a special notation: given $\alpha \in \mathcal{F}$, $x \in \mathcal{X}$, and $k \in \mathcal{K}$, if there exists $m \in \alpha$ that flows at x with version k, the term $\alpha_x(k)$ represents the value $\nu(m)$ stored in it (or if no such m exists, $\alpha_x(k)$ is undefined). Equation (3) guarantees that if $\alpha_x(k)$ is defined, it is unique. Formally, the following holds for all messages $m \in \alpha$:

$$\chi(m) = x \wedge \kappa(m) = k \Rightarrow \alpha_x(k) \stackrel{def}{=} \nu(m) \,. \tag{4}$$

Note that neither of the equations discussed so far places any constraints on messages at different locations. A flow is said to be *consistent* if the following stronger requirement holds:

$$\kappa(m) = \kappa(m') \Rightarrow \nu(m) = \nu(m') \,. \tag{5}$$

For consistent α, we can further shorten our simplified notation introduced above: instead of $\alpha_x(k)$, we write just $\alpha(k)$.

Consistency is a relatively strong property; it does not apply to most flows; normally, only output flows in objects that implement some variant of distributed agreement are consistent. In our language, consistent flows are produced using a mechanism called *aggregation* (discussed in Section 2.4.3).

A much more common property is for a flow to be *weakly monotonic*; in such flows, messages with higher versions, at the same location, must have larger values, as defined below:

$$\chi(m) = \chi(m') \wedge \kappa(m) \leq \kappa(m') \Rightarrow \nu(m) \leq \nu(m') \,. \tag{6}$$

Such flows are also called *weakly increasing*. Flows that are weakly monotonic with respect to the opposite order ($\geq$) are called *weakly decreasing*. If a flow is weakly increasing and weakly decreasing, it is *constant*. This is defined as follows:

$$\chi(m) = \chi(m') \Rightarrow \nu(m) = \nu(m') \,. \tag{7}$$

A flow is *strongly monotonic* (or *monotonic*) if monotonicity holds across locations, as defined below; terms (*strongly*) *increasing* and (*strongly*) *decreasing* are defined accordingly:

$$\kappa(m) \leq \kappa(m') \Rightarrow \nu(m) \leq \nu(m') \,. \tag{8}$$

It's easy to verify that a monotonic flow is always consistent. As we explain later, monotonicity is the single most essential property in our model, and it is a universal tool for reasoning about the behaviors of the constructed protocols.

Neither of the flows *wants* or *holds* in Example 1 is monotonic (not even weakly) nor consistent, but as we'll see in Section 2.3, the implementation of $lock$ in our flow language involves an internal consistent flow to represent global state.

2.3 Programs

In this section, we introduce the formal language syntax. The essential rules are listed on Figure 5. For the sake of brevity, we omit rules for defining new data structures and arithmetic operators; these are similar as in Java. Our first example, the code of a distributed locking protocol, is shown on Figure 6. The language features it uses, and the embedded object $stable\_elect$, are gradually introduced in the following sections.

The definition of a new LO (syntax rule R10 on Figure 5) in its default form (**object** i {e_1;...e_k; c}) consists of the name of the object (i), definitions of all endpoints it exposes as its external interface (e_j), and code (c), which could declare internal flows (v_j in rule R02). Endpoints and internal flows are the analogues of public methods and private fields of objects in Java. Endpoint definition (rule R04), like a method,

includes its name (i), signature (s), and code (c), and the definition of an internal flow (rules R17 and R05) includes the type of messages in the flow (t), optionally preceded with its *properties*, followed by name (i) and optionally, initial value (y). Properties (rule R11) mark flows as weakly or strongly increasing (**up** or **s-up**), weakly or strongly decreasing (**down** or **s-down**), consistent (**same**), or constant (**const**). Endpoint signatures (rule R14) resemble Java method signatures; they consist of the lists of all input (f_k^i) and output flows (f_k^o).

In the second variant, the object definition doesn't include the explicit endpoint declarations, but rather has the endpoint signature (s) following the object's name. In this variant, the object is assumed to expose a single unnamed endpoint with this signature; this is the case for the $lock$ object on Figure 6. This construction is conceptually similar to a .NET *delegate*.

Unlike the body of a Java method, the code (c) embedded in the endpoint definition (R04) doesn't execute just once; it runs continuously, from the moment *some* endpoint instance is connected, until the moment when *every* endpoint instance is disconnected. As new messages appear in the input flows, the endpoint code may produce messages in either the output flows, or in the object's internal flows. The code (c) should not be viewed as a sequential set of operations, but rather as a set of flow dependencies that execute *concurrently*, *continuously*, and as explained later, in a *coordinated* fashion.

Partitioning code among endpoint declarations allows the object's proxies to behave differently depending on the functional roles they play. In the LO model, these roles depend on which endpoint instances are connected. For example, an object with replicated state might expose two endpoints, $client$ for accessing the state, and $replica$ for providing the storage. Code embedded in the body of endpoint $replica$ would then run only among proxies that are hosted on servers, and have their $replica$ endpoint instances connected (Figure 7). Code outside of endpoint definitions (c in rule R10) runs on proxies that have at least one instance of any endpoint connected; this is also the case for code in lines 02-05 on Figure 6.

Code consists mostly of *dependencies* (rule R03), which resemble assignments in Java. Each dependency defines a set of flows in terms of other flows. In the example on Figure 6, in line 04 internal flow $owner$ is defined as the output flow of the embedded object $stable\_elect$, and in line 05, the output flow $holds$ is defined in terms of flow $owner$. As mentioned earlier, our code is not sequential; all dependencies thus run in parallel, generating new messages in the *dependent* flows (those listed on the left side of the assignment operator := in rule R03) from those they depend on (on the right side of :=). In our example, whenever a message appears at the output of $stable\_elect$ it is copied to the internal flow $owner$ (line 04). This, in turn, causes expression in line 05 to recompute, and a new message flows in $holds$. Computation is event-driven, much as in rule-based systems using the Rête algorithm [15].

As suggested above, dependencies can refer to embedded objects (line 04). The other form $i_1,...i_k$:= $[h]\ i[.i'](x_1,...x_k)$

R01: aggr. attrib.	a ::= **unordered** | **incomplete** | **uncoordinated**
R02: code	c ::= v_1 ; ... v_j ; l_1 ; ... l_k ;
R03: dependency	d ::= i := x | i_1 , ... i_k := $[h]\ i[.i'](x_1$, ... $x_k)$
R04: endpoint	e ::= **endpoint** $i\ s$ { c }
R05: flow	f ::= p_1 ... $p_k\ t\ i$
R06: aggr. oper.	g ::= **or** | **and** | **min** | **max** | **add** | **mul** | **union** | **intersect**
R07: hier. attrib.	h ::= **independently**
R08: line of code	l ::= { c } | d | **where** (x) c_1 [**elsewhere** c_2]
R09: modifier	m::= **other** | **fresh** | **some**
R10: object	o ::= **object** i { e_1 ; ... e_k ; c } | **object** $i\ s$ { c }
R11: property	p ::= **same** | **const** | **up** | **down** | **s-up** | **s-down**
R12: constant	q ::= b | n | { q_1 , ... q_k} | $\varnothing$ | (q_1 , ... q_k) | **id**
R13: infix oper.	r ::= $=$ | $\neq$ | $\leq$ | $<$ | $>$ | $\geq$ | $\wedge$ | $\vee$ | $+$ | $-$ | $*$ | $/$ | $\cup$ | $\cap$ | $\setminus$ | $\in$ | $\notin$ | $\subseteq$ | $\subset$ | $\supseteq$ | $\supset$
R14: signature	s ::= (f_1^i , ... f_k^i) : f_1^o , ... f_k^o
R15: type	t ::= **bool** | **int** | (t_1 , ... t_k) | { t }
R16: unary oper.	u ::= $\neg$ | $-$
R17: variable	v ::= f [:= q] | **object** [h] $i\ i'$
R18: expression	x ::= [m] i | q | $u\ x$ | $x\ r\ x$ | (x_1 , ... x_k) | x @ n | { x_1 , ... x_k} | x' | [x_1 , x_2] | (x) | g (x_1 , ... x_k) | a_1 ... $a_k\ g\ i$ | **singleton**

Figure 5. The syntax rules of our distributed data flow language expressed in a notation similar to BNF. Keywords are bold, sans serif, and in blue, non-alphanumeric terminals are bold and in red, and non-terminals are italic. Optional occurrence of z is written as "[z]". Subscripts and superscripts are not parts of non-terminal names; they're used to distinguish between distinct occurrences. Repetition of a non-terminal z is expressed as "$z_1 ... z_k$" (or if y is a delimiter: "$z_1 y ... z_k$"). We omit rules for identifiers (i), numbers (n), Booleans (b), and rules for defining custom (ordinary) data structures and arithmetic operators (these are similar to rules in Java).

```
01: object lock(bool wants) : bool holds { // endpoint sign.
02:     same int owner; // an internal consistent flow
03:     where (wants) // this determines who runs line 04
04:         owner := stable_elect(id); // an embedded object
05:     holds := wants ∧ (owner = id); } // flow dependency
```

Figure 6. The code of a distributed locking LO expressed in our language. Via an unnamed endpoint, an object named *lock* consumes a Boolean flow $wants$, and outputs a Boolean flow $holds$ (line 01). It uses a consistent internal flow $owner$ (line 02) to store the identifier of the node that holds the lock. All nodes that would like to acquire the lock (line 03) submit identifiers to the embedded object $stable\_elect$ (line 04; for the code consult Figure 17). If the local id matches that of the leader, the proxy holds the lock (line 05). The result is stable until the lock owner quits (this is discussed in Section 2.4.3).

in rule R03 is like a method call: i is the embedded object's name, i' is the optional name of one of i's named endpoints if it has any (otherwise, we assume the unnamed endpoint is selected), flows i_j mirror i's outputs and i's inputs are fed from expressions x_j. Each use of such dependency declares a separate embedded object, a single proxy of which is embedded in each proxy of the object being defined (Figure 8). Pairs of proxies communicate using their endpoint instances. The embedding object can programmatically connect or disconnect the endpoint instance it uses to talk to the embedded object, effectively activating or deactivating code in the latter (recall our earlier discussion, and the example on Figure 7). For example, the conditional **where** statement in line 03 activates/deactivates the embedded proxy of $stable\_elect$ based on values in flow $wants$, so that only proxies that intend to grab the lock are participating in the leader election protocol.

Sometimes, one may wish to use multiple instances of the same type of object (e.g., multiple instances of $stable\_elect$) for different purposes, or one may wish different sections of the code to interact with the embedded object using different endpoints. This is achieved by declaring embedded objects in the variables section (rule R17), via pattern **object** $[\,h\,]\,i\,i''$, where i identifies the object type (e.g., $stable\_elect$), and i'' is an alias that we'll use to refer to a particular instance of it. When i'' is substituted for i in $i_1,\ldots i_k$:= $[h]\ i[.i'](x_1,\ldots x_k)$, the pattern doesn't declare a new embedded object instance; it connects to the one we declared earlier with **object** $[\,h\,]\,i\,i''$.

For each input and output flow defined in endpoint signature (rule R14), and for each internal flow defined in the code (rule R02), every proxy keeps one message queue (Figure 8, Figure 9). The program on Figure 6 creates six such queues: two for flows $wants$ and $holds$ declared in $lock$'s endpoint signature (line 01), two for flows $candidate$ and $leader$ created by the recursively embedded proxy of $stable\_elect$, one for the embedded flow $owner$, and one for **id** (line 05). Dependencies are generally implemented by pulling messages from some queues, transforming them and storing the results as messages in other queues. Often, this happens locally, but sometimes it involves coordination with other proxies.

As mentioned earlier, computations are event-driven. One can think of each running object as a Petri net [39], in which flows play the roles of locations, and dependencies the roles of transitions. Every time a new message appears in some of the flows that serve as sources of data for a dependency (the dependency's *preset*), this triggers calculations and produces a message in the dependent flow (the dependency's *postset*). One can also use the Petri net analogy at a more mechanical level, where the individual message queues play the roles of locations, and the computations play the roles of transitions.

The meaning of conditional **where** (x) c **elsewhere** c' is that code c is locally activated (and c' deactivated) at a proxy as soon as a message with value $true$ flows in the local queue that is a part of the flow represented by x, and likewise, c is deactivated (and c' activated) when a message with the $false$ value appears in that queue. Initially, neither c nor c' is active. In case of embedded objects, **where** connects or disconnects endpoint instances. In dependencies, **where** suspends or resumes message exchange with queues corresponding to

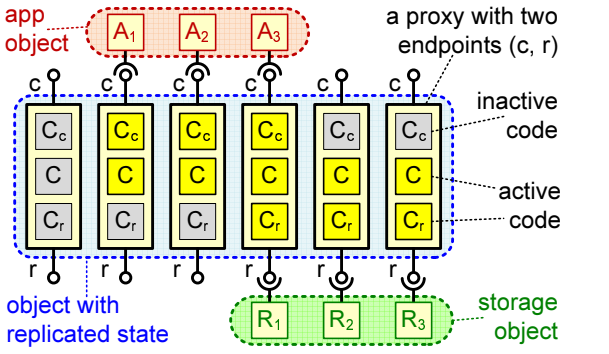

Figure 7. Code C_c in the body of endpoint c (rule R04) runs on proxies that have their instances of c connected. Likewise, C_r in the body of endpoint r runs on proxies that have their instances of r connected. Code C (R10) runs in either case.

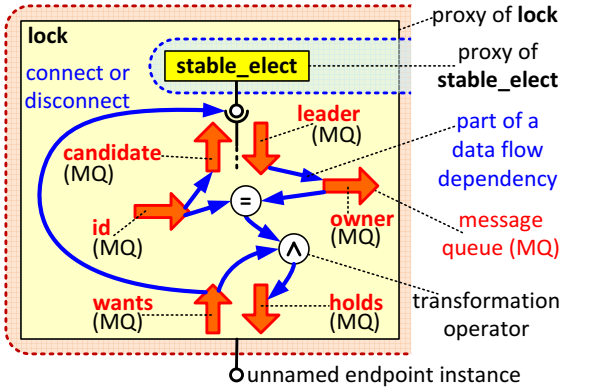

Figure 8. Dependency in line 04 in Figure 6 embeds proxies of $stable\_elect$ in proxies of $lock$, binds its input to flow id, and routes its output into $owner$. Values in $holds$ are generated from those in id, $wants$, and $owner$ (line 05). Values in $wants$ activate or deactivate the connection to $stable\_elect$.

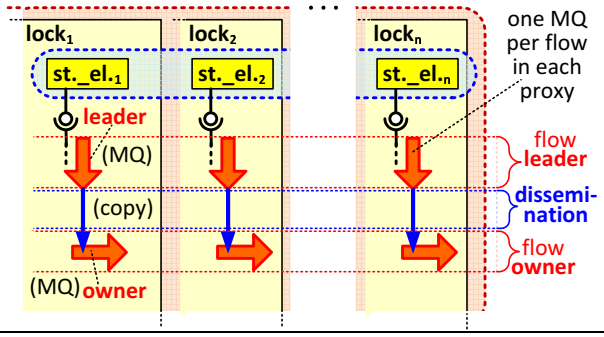

Figure 9. For each flow $\alpha \in \mathcal{F}$ in an object definition, the proxy at each location x maintains a local message queue for messages $\{m \in \alpha \mid \chi(m) = x\}$. Dependencies move messages between queues and transform them in-flight; this is usually done concurrently and independently on all proxies.

flows used in c or c'. For example, the **where** clause in line 03 on Figure 6 controls (locally at each proxy) whether messages can be put in the local queue $owner$ or pulled from **id**.

Flow **id** (line 05) is an example of a built-in constant flow. Each message queue of a constant flow contains only a single message. In case of **id**, the value of this message is a globally unique numeric identifier of the proxy's location. Constant flows participate in computation just like any other flows; for example, whenever $lock$ internally connects to $stable\_elect$, it copies the (single) message from **id** into $candidate$. Every occurrence of a numeric or Boolean constant (rule R12) also defines a constant flow; in this case, messages in all queues carry the same value. The language also supports tuple constants of the form $(q_1,...q_k)$, and set constants: $\varnothing$, $\{q_1,...q_k\}$.

The built-in message types (rule R15) include Integer (**int**) and Boolean (**bool**); these can be further combined into tuple types (denoted as $(t_1,...t_k)$), and set types (denoted as $\{t\}$). Supporting custom, user-defined types would be helpful, but to keep it simple, we limit ourselves to the built-in types. The integer set type **{int}** is particularly useful; it enables batched processing (we discuss one example of this in Section 2.4.1).

Standard types come with logical ($\neg$, $\wedge$, and $\vee$), comparison ($=$, $\neq$, $\leq$, $<$, $\geq$, and $>$), arithmetic ($+$, $-$, $*$, and $/$), and set operators ($\cup$, $\cap$, $\setminus$, $'$, $\in$, $\notin$, $\subseteq$, $\subset$, $\supseteq$, $\supset$). One can construct sets $\{x_1, ...x_k\}$ and tuples $(x_1, ...x_k)$ from elements, remove elements from tuples ($x@n$ is the element at n-th position), and define ranges of numbers ($[x_1,x_2]$ is only supported for x_1, x_2 numeric; it represents the set $\{n \in \mathbb{N} \mid n_1 \leq n \leq n_2\}$, where n_1, n_2 are the values of x_1, x_2. Finally, one can apply *aggregation operators* in prefix notation $g(x_1, ...x_k)$, where g can be any of: **min**, **max**, **or** (alternative), **and** (conjunction), **add** (sum), **mul** (product), **union**, or **intersect** (intersection). The way these are interpreted is discussed in Section 2.4.3.

The remaining language constructs are described later. To conclude this section, we need to make a comment about the peculiar set arithmetic used in our language. The representation of sets could, in general, consume a lot of space, but our language is designed to support high-performance protocols that might need to run with limited network bandwidth. This means that when transmitted over the network or aggregated across sets of machines, set values may need to be truncated.

Accordingly, at runtime each set value A is actually represented as a pair (A^+, A^-), where A^+ is the set of elements that definitely belong to A, and A^- is the set of elements that definitely do not belong to A; for all other elements, it is undefined. Set operations are then defined to preserve as much information as possible; for example, in the union $A \cup B$, we can only guarantee that elements $A^+ \cup B^+$ are in the result. Formal definitions of the four basic operators are as follows:

$$(A^+, A^-) \cup (B^+, B^-) = (A^+ \cup B^+, A^- \cap B^-)\ , \quad (9)$$

$$(A^+, A^-) \cap (B^+, B^-) = (A^+ \cap B^+, A^- \cup B^-)\ , \quad (10)$$

$$(A^+, A^-) \setminus (B^+, B^-) = (A^+ \cap B^-, A^- \cup B^+)\ , \quad (11)$$

$$(A^+, A^-)' = (A^-, A^+)\ . \quad (12)$$

```
01: object repair(int addr, {int} recv) : {(int,{int})} fwd {
02:     fwd := { ( other addr, recv \ other recv ) };
03: }
```

Figure 10. A simple form of loss recovery: $recv$ carries sets of identifiers of network packets locally received at the given location (line 01) into the object, and fwd carries forwarding requests out of it. Each node forwards to a certain other node packets that are available locally, but not remotely (line 02).

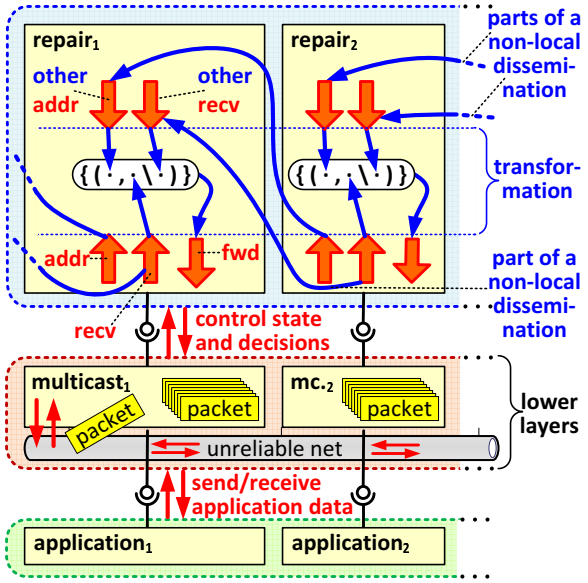

Figure 11. The loss recovery object $repair$ from Figure 10 only deals with control decisions: it detects packet losses and issues forward requests; another object $multicast$ that interfaces it handles the actual transmissions, caching, interacts with the application, and report its local state to $repair$. For the two non-local disseminations **other** $addr$ and **other** $recv$ in the $repair$ object, each proxy pulls messages into its local queue from a remote queue maintained by some other proxy.

In particular, values of type **{int}** are represented as tuples of the form $(a, (a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k), b)$. Whether n is in the set is defined if $a \leq n \leq b$ (and undefined otherwise); if it's defined, n is in the set iff $\exists_{1 \leq i \leq k}\ a_i \leq n \leq b_i$. In our simulations, this representation proved to have a fairly small CPU and space overhead [38]; we discuss it in Section 2.4.1.

2.4 Dependencies

Dependencies are the basic building blocks in our language. We distinguish four kinds of these: *disseminations*, *transformations*, *aggregations*, and *distributions*.

2.4.1 Disseminations

This is the simplest of dependencies. Flow $\beta \in \mathcal{F}$ is a *dissemination* of $\alpha \in \mathcal{F}$ if each value appearing in β has appeared previously in α, at the same or some other location:

$$\forall_{m \in \beta}\ \exists_{m' \in \alpha}\ \nu(m') = \nu(m) \wedge \tau(m') < \tau(m)\ . \quad (13)$$

Message m in this equation is said to *depend* on message m' that provided the value. Dissemination β is *local* if the latter flows at the same location (that is, if equation (13) still holds after appending to it the extra condition "$\wedge\chi(m') = \chi(m)$").

In our $lock$ example (Figure 6), *owner* is a local dissemination of the output flow *leader* of the embedded object *stable_elect*. As mentioned earlier, we implement this by locally pulling messages from the $leader$ queue, and copying them to the $owner$ queue, independently on each proxy (Figure 9).

To more accurately describe the way dependencies work, we characterize them through their *membership* and *selector* functions. In case of dissemination, memberships form family of partial functions $\mu_x : \mathcal{K} \to \mathcal{X}$, and selectors are partial functions $\sigma_x : \mathcal{K} \to \mathcal{K}$, for each $x \in \mathcal{X}$, where $\text{dom}(\mu_x) = \text{dom}(\sigma_x) = \{k \in \mathcal{K} \mid \exists_{m\in\beta}\ \chi(m) = x \wedge \kappa(m) = k\}$. For each message $m \in \beta$, if its location is $x = \chi(m)$ and version is $k = \kappa(m)$, then the location of the original message $m' \in \alpha$ that m depends on is $\mu_x(k)$, and its version is $\sigma_x(k)$. Using this new notation, one can then express β as follows:

$$\beta_x(k) = \alpha_{\mu_x(k)}(\sigma_x(k))\ . \tag{14}$$

Note how subscript $\mu_x(k)$ in $\alpha_\bullet(\bullet)$ selects location, and the argument $\sigma_x(k)$ selects version. The above notation stresses the functional nature of dependencies, and helps distinguish their different flavors, e.g., in a local dependency $\mu_x(k) = x$.

Dissemination is *in-order* if for every two messages in β that flow at the same location, if one has larger version than the other, then the versions of messages in α they depend on are in a similar relation; formally, this requires that selectors are monotonic, i.e., $\forall_{x\in\mathcal{X};k,k'\in\mathcal{K}}\ k \le k' \Rightarrow \sigma_x(k) \le \sigma_x(k')$. Intuitively, it means that when pulling messages, each queue tries to keep only the fresh ones, and ignore the old ones. In our language, disseminations are in-order by default.

Now, let's analyze an important example of non-local dissemination, which additionally illustrates batch processing.

Example 2. Object $repair$ (Figure 10) implements a simple form of multicast loss recovery: pairs of proxies compare sets of identifiers of all network packets received locally, and whenever one of them finds that it has packets that a certain other node is missing, it generates a forwarding request. Our object doesn't deal with the physical network transmissions; we assume it is connected to another object that performs all the low-level tasks. Object $repair$ implements only core decision logic: it detects when losses occur, and decides which nodes should forward data to which other nodes (Figure 11). This general pattern of use applies to all protocols presented in this paper (our platform [1] supports such compositions).

Flow $addr$ carries into each proxy of $repair$ the network address at which its local node can receive network packets forwarded by other nodes. Flow $recv$ of the integer set type {int} carries into it sets of identifiers of all packets received locally. For example, if at time t, the proxy of $multicast$ at node x receives packets with identifiers 28..29 from the network, and it has previously received packets with identifiers

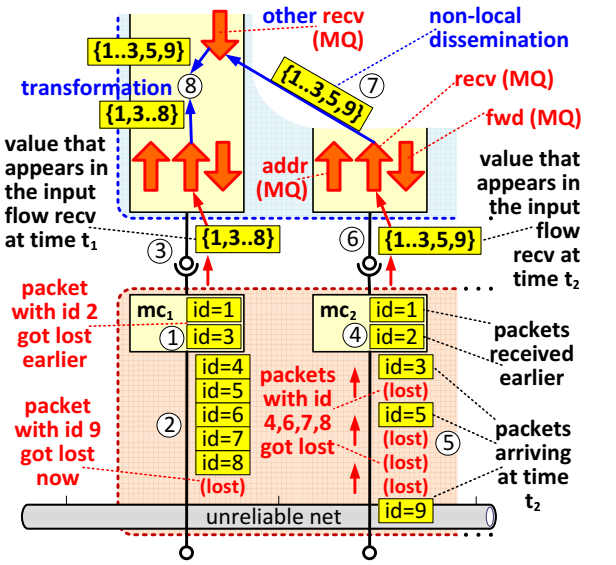

Figure 12. Batched processing with set arithmetic in protocol $repair$: (1) before time t_1, node 1 received packets with identifiers 1 and 3 and is caching them in its local proxy mc_1 of the multicast object; packet with identifier 2 never arrived; (2) at time t_1, a batch of packets with id from 4 through 8 is received; (3) proxy mc_1 now decides to report the new status to the local proxy of $repair$; a message carrying a single set value $\{1, 3..8\}$ flows in $recv$ at that node around time t_1; it is received by the proxy of $repair$ and put into its input queue; (4) likewise, before time t_2 node 2 received packets 1..2 and is caching them; (5) at time t_2 it gets new packets 3, 5 and 9; packets 4 and 6..8 got lost; (6) at time t_2, proxy mc_2 reports its updated status to its local proxy of $repair$, sending to it a single set value $\{1..3, 5, 9\}$; (7) the latter value is forwarded to node 1, and appears in its local queue **other** $recv$; (8) eventually, this triggers local computation in line 02, the two values are substracted, and local proxy $repair_1$ requests that packets with ids in the set $\{1, 3..8\} \setminus \{1..3, 5, 9\} = \{4, 6..8\}$ be forwarded to node 2; this appears in the output flow fwd as a value of the form $fwd_1(k) = \{(addr_2(k'), \{4, 6..8\})\}$.

1..25, then the proxy $multicast_x$ will send to its locally connected proxy $repair_x$ a message with value $\{1..25, 28..29\}$; formally, $\exists_{k\in\mathcal{K}}\ (x, t, k, \{1..25, 28..29\}) \in recv$ (Figure 12).

Flow fwd carries sets of forwarding requests. Its values are sets of pairs (a, S), in which a is the address of the node that should receive the forwarded packets, and S is the set of identifiers of all those packets. For example, a request that x forward packets 1..5 to $lion$ and packets 8..9 to $tiger$ would be modeled as $fwd_x(k) = \{(lion, \{1..5\}), (tiger, \{8..9\})\}$. Proxies of $repair$ produce such values in line 02, by comparing the values received in $recv$ locally and elsewhere. ■

Notice the use of **other** $recv$ in line 02 on Figure 10. In general, the construct **other** i, where i is a flow name, represents what one might think of as a "shifted" flow: each proxy fetches a value appearing in flow i at its *neighboring* proxy;

thus, the values of **other** $recv$ at a given proxy will be sets of identifiers of messages received by its neighbor. The neighboring proxy is the same for each occurrence of **other** i. It can change over time, but it always does so atomically with respect to computation performed by the proxy. Neighboring relationships are asymmetric. The runtime ensures that they are configured so that if we represent them as a graph, at any point in time one can travel from any point in such graph to any other point; in other words, information in each proxy eventually affects information in every other proxy, directly or indirectly. This can be implemented in a variety of ways, e.g., by organizing the proxies into token rings, where each proxy sets the successor on the ring as its neighbor, strongly connected trees, where each proxy periodically switches between neighbors in the tree, or randomized gossip protocols.

We'd like to highlight one important aspect of this example: thanks to our use of set arithmetic, in a single step in the computation information about multiple application events can be processed simultaneously, in batch mode (Figure 12); for example, if one proxy received messages with identifiers 1..1000, and its neighbor 1..950, expression $recv \setminus$ **other** $recv$ in line 02 yields a forwarding request for packets $951..1000$, all at once. This way, control traffic running at rates as low as a few protocol rounds/second can potentially support protocols that handle thousands of application events/second [38].

Let's try to estimate the gain. Suppose packet loss occurs at random with probability p, and that set values are encoded as numeric ranges, as explained at the end of Section 2.3 (we used this encoding successfully in our simulations [38], and in our earlier high-performance multicast implementations). Let $\lambda(p)$ be the average number of consecutive packets that are either all lost or all received by a node; it's easy to check that $\lambda(p) = (p \cdot (1-p))^{-1} - 2$, and if p is small, $\lambda(p) \approx 1/p$. Let the maximum size that can be occupied by each set value in memory or in a network packet be S bits, and suppose that besides a tiny header that encodes the first identifier that is or isn't in the set, almost all of the remaining bits are used to store a list of B-bit numbers; each of these is the length of a single series of consecutive identifiers that are (or not) in the encoded set value. For example in value $\{1..25, 28..29\}$, the header would specify that the first identifier is 1, and it is in the set, and it would be followed by three B-bit numbers: 25, 2, 2, indicating that starting with 1, there are 25 consecutive identifiers in the set (1..25), then 2 not in the set (26..27), and then again 2 in the set (28..29). With S bits, one can encode information about $\approx S/B$ series of consecutive identifiers (ignoring the header), each series of length $\lambda(p)$ on average. The total number of identifiers for which the information of whether they are (or not) in the set can be encoded in a single set value is $\approx S/(Bp)$. For example, if loss rate is $p = 1\%$, we allow 4 KB space per value, and $B = 16$ bits, each value could carry information about ≈ 2000 identifiers on average. If the multicast rate is at the order of 2000 packets/s or less,

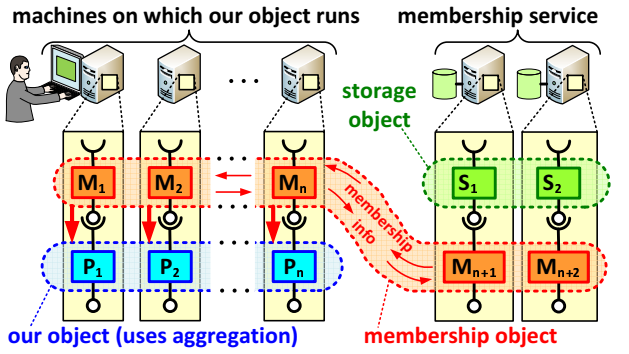

Figure 13. An object that uses aggregation is supported by an external membership service (MS), which provides all the object's proxies with consistent membership views; these are used to self-organize and form structures such as token rings.

it would suffice for computations in our language to fire (and for set values to be disseminated) as rarely as once a second.

2.4.2 Transformations

Flow $\beta \in \mathcal{F}$ is a *transformation* over flows $\alpha^1, \ldots \alpha^n \in \mathcal{F}$ if there exists an n-argument function $\Psi : \mathcal{V}^n \rightarrow \mathcal{V}$ such that for each message $m \in \beta$ in this flow, its value $\nu(m)$ can be represented as a result of applying function Ψ to a list of values that appeared in flows α^i (a single value from each):

$$\forall_{m \in \beta}\, \exists_{m'_1 \in \alpha^1} \ldots \exists_{m'_n \in \alpha^n}\ (\forall_i\, \tau(m'_i) < \tau(m)) \wedge \ldots \\ \cdots \wedge \nu(m) = \Psi(\nu(m'_1), \ldots \nu(m'_n))\ . \quad (15)$$

Message m is said to *depend* on all m'_i used in the equation. Transformation is *local* if m and m'_i appear at the same location (if we can append $\wedge \forall_i\, \chi(m'_i) = \chi(m)$ in equation (15)). Every occurrence of a logical, arithmetic, comparison, or aggregation operator in an expression defines a local transformation (except for the pattern $a_1 \ldots a_k\ g\ i$ in rule R18, which is an aggregation; this is discussed in the following section).

In example $lock$, in line 05, we define $holds$ as a transformation on $wants$, $owner$, and **id**, with $\Psi(v_1, v_2, v_3) \triangleq true$ if $v_1 \wedge (v_2 = v_3)$, else $false$. In example $repair$, in line 02, we define fwd as a transformation on **other** $addr$, $recv$, and **other** $recv$, with $\Psi(v_1, v_2, v_3) \triangleq \{(v_1, v_2 \setminus v_3)\}$ (Figure 11).

One can characterize a transformation via its membership functions $\mu^i_x : \mathcal{K} \rightarrow \mathcal{X}$, and selector functions $\sigma^i_x : \mathcal{K} \rightarrow \mathcal{K}$, for $1 \leq i \leq n$, $x \in \mathcal{X}$, as follows (the domains and meaning of μ^i_x and σ^i_x are defined just as we did it for dissemination):

$$\beta_x(k) = \Psi(\ldots,\ \alpha^i_{\mu^i_x(k)}(\sigma^i_x(k)), \ldots)\ . \quad (16)$$

Transformation is *in-order* if all selectors are monotonic. By default, all transformations in our language are local and in-order: $\mu^i_x = \mu^i_{x'}$ for $x \neq x'$ and $\sigma^i_x(k) \leq \sigma^i_x(k')$ for $k \leq k'$.

2.4.3 Aggregations

Aggregation is the core concept in our language; it allows us to achieve strong semantics. Unlike dissemination and trans-

formation, aggregation generally requires proxies to cooperate. This can be facilitated by an external membership service (MS): proxies can use it to self-organize into a group, and perform aggregations together (Figure 13, 14, 15, and 16).

Flow β is an *aggregation* on flow α if every value flowing in β can be represented as a result of applying some associative commutative binary operator $\otimes : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ to some set of values that have previously appeared in α:

$$\forall_{m\in\beta}\ \exists_{S\subseteq\alpha}\ |S| < \infty \wedge (\forall_{m'\in S}\ \tau(m') < \tau(m)) \wedge \ldots \\ \ldots \wedge \nu(m) = \bigotimes_{m'\in S} \nu(m')\ . \quad (17)$$

The standard aggregation operators listed in rule R06 are all associative and commutative, and can be substituted for g in the pattern i_1 := $a_1 \ldots a_k\ g\ i_2$ to define i_1 as an aggregation over i_2. Attributes a_i are used to customize the way in which aggregation is performed; this is discussed below.

As with disseminations and transformations, we can characterize aggregations by their memberships $\mu_x : \mathcal{K} \to \mathbb{P}(\mathcal{X})$ and selectors $\sigma_x^y : \mathcal{K} \to \mathcal{K}$, for $x, y \in \mathcal{X}$. Each membership μ_x, given a version $k \in \mathcal{K}(\beta)$ of some messages in β, selects a set of locations in α, and for each location $y \in \mu_x(k)$, the selector $\sigma_x^y(k)$ further specifies the version of a message that this location contributes to the aggregation (Figure 15):

$$\beta_x(k) = \bigotimes_{y\in\mu_x(k)} \alpha_y(\sigma_x^y(k))\ . \quad (18)$$

Aggregation is *in-order* if the selectors σ_x^y are monotonic. It is *coordinated* if memberships and selectors are identical at different locations x, i.e., $\mu_x = \mu_{x'} \wedge \sigma_x^y = \sigma_{x'}^y$ for $x \neq x'$; intuitively, this means that when aggregating values with the same version, different proxies select the same locations and the same versions at those locations in order to calculate their results. Finally, aggregation is *complete* if every permanent location of α eventually starts to contribute its value to every subsequent aggregation in β. By default, aggregations in our language have all these properties unless explicitly annotated as **unordered**, **uncoodinated**, or **incomplete** (rule R01).

To root things in physical reality, let's see how to implement such aggregation if nodes self-organize as a token ring with the help of an external membership service (Figure 16). Aggregations are performed in rounds; a token is circulated with the partial result that each node contributes to by merging the result with its local value, then passing it further on, and the final aggregation results are tagged with versions of the form $k = (i, j)$, where i is the number of the membership view, and j is the number of the token round in the view[5]. The in-order property is achieved if nodes contribute the latest local values they got. Coordination is achieved because aggregation results are collected by a single node and handed

[5] Full details of the runtime infrastructure and distributed protocols used to support aggregation are beyond the scope of this paper; they'll be presented in a journal paper. The key concepts have been implemented and tested [38].

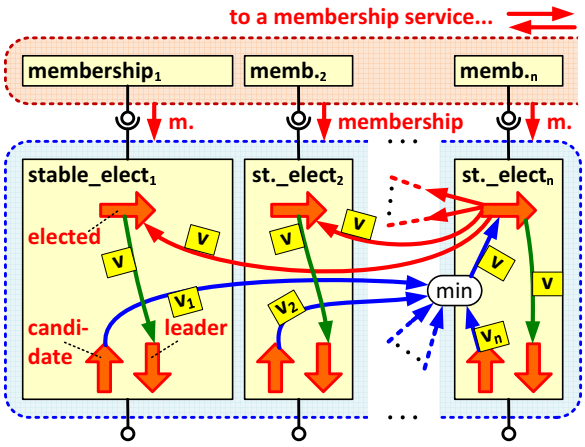

Figure 14. A group of proxies computing aggregation in the protocol from Figure 17. In each aggregation round, sets of values v_i that appear in flow $candidate$ are aggregated into a single value $v = \min_{1\le i\le n} v_i$, which emerges at the leader node. The result is disseminated to all proxies. Proxies self-organize into a token ring or a similar structure with the help of *membership views* (provided by the $membership$ object).

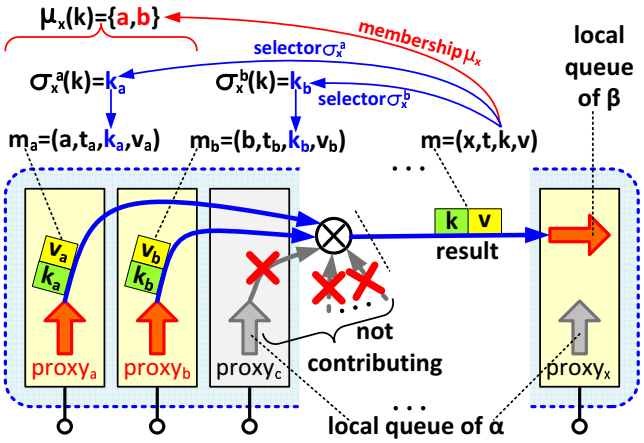

Figure 15. Membership $\mu_x(k)$ selects locations that participate in aggregating values for messages with version k at x, and selectors $\sigma_x^y(k)$ further specify the versions of messages that each location $y \in \mu_x(k)$ contributes to the aggregations.

out to everyone. Completeness follows from the fact that the membership view specifies who precisely is in the ring, and permanent locations remain in the membership view forever.

It's not hard to see that coordinated aggregation is consistent; our aggregations are thus consistent by default. In fact, many classes of aggregations are easily proven to be strongly monotonic. To explain this, we need a few extra definitions.

An aggregation operator $\otimes$ is *monotonic* if it satisfies the first of the following two equations, and it is a *lower bound* if it additionally satisfies the second one:

$$\forall_{v_1,v_2,v_3\in\mathcal{V}}\ v_1 \le v_2 \Rightarrow v_1 \otimes v_3 \le v_2 \otimes v_3\ , \quad (19)$$

$$\forall_{v_1,v_2\in\mathcal{V}}\ v_1 \otimes v_2 \le v_1\ . \quad (20)$$

All standard aggregation operators in our language (rule R06) are monotonic, and **and**, **min**, and **intersect** are lower bounds.

A coordinated aggregation β on α is *guarded* if each time a new node joins the subsequent aggregation, it ensures that the value it will contribute is not smaller than either partial or full result of the ongoing or the immediately preceding aggregation. In the scenario on Figure 16, it means that each time a node is about to place its local value into the token or merge its local value with the one in the token (steps 1, 4, 7), it has to double-check that the local value is no smaller than the result of the preceding aggregation; if it is, the node can't contribute its local value now and has to wait until it grows.

Formally, for all pairs of subsequent versions $k < k'$ in β (i.e., such that $\neg\exists_{k''}\ k < k'' < k'$), and for every location y joining the aggregation at version k' ($y \in \mu_x(k') \setminus \mu_x(k)$):

$$\exists_{S\subseteq\mu_x(k)} \left(\alpha_y(\sigma_x^y(k')) \geq \bigotimes_{z\in S} \alpha_z(\sigma_x^y(k))\right) . \qquad (21)$$

Aggregation in our language is not guarded by default, but it can be easily made such. Recall that conditional **where** (x) c locally deactivates dependencies in c at a given proxy when condition x at this proxy locally evaluates to $false$, that is, as soon as a message carrying value $false$ appears in the local queue of the flow that is carrying the results of expression x. In case of aggregations in c, being locally deactivated means that the given proxy contributes values from its local queues only at the times when condition x locally holds. Earlier, we stated that aggregation is complete by default, but **where** (x) creates an exception: aggregation by default runs among all proxies in the group *except* those explicitly excluded from it by the **where** clause. Now, consider the following pattern:

$$\textbf{where}\ (\textbf{fresh}\ \beta \wedge \beta \leq \alpha)\ \beta := \otimes\ \alpha;\ . \qquad (22)$$

Expression of the form **fresh** β locally evaluates to true on a given proxy when the proxy is sure it has the latest possible value of β. This feature is easy to implement if aggregation is performed using a token ring protocol under the control of a membership service (as discussed earlier); as soon as a proxy joins the ring, all tokens carrying partial results pass through it, so it always knows which version was the latest[6].

At this point, we need to make one important comment: in general, it is possible that all nodes crash or simultaneously leave the protocol, and later the protocol resumes with a new set of entirely different nodes. In such situations, of course, there's no way that the new nodes can learn the results of past aggregations, or even determine what was the latest version. Thus, aggregation couldn't be guarded (or even consistent) if the flow could span such events. Accordingly, when such an event occurs, we assume that the existing flow ends, and a new instance of it begins; we say that the flow has *rebooted*[7].

[6] Technical details are fairly straightforward; the approach has been outlined in our technical report [38]. More details will be provided in a journal paper.

[7] This appears to be a standard assumption in most distributed protocols that support dynamic membership, although often it is not explicitly verbalized.

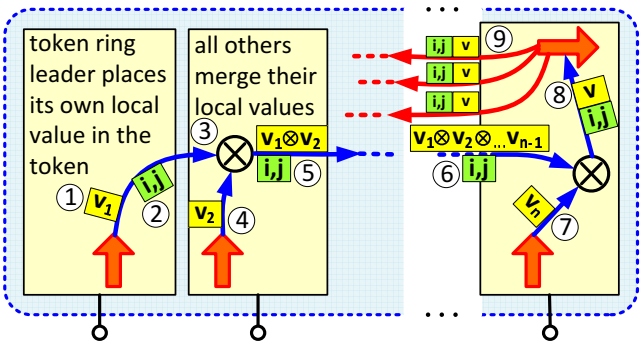

Figure 16. Aggregation using a token ring protocol: (1) the ring leader puts its local value v_1 in the token; (2) it also puts in it version (i, j), where i is the number of the membership view in which this aggregation round is happening, and j is the number of the aggregation round within the view; (3) the combo $(i, j), v_1$ arrives at the second node in the ring; (4) the node pulls its local value; (5) values are merged into $v_1 \otimes v_2$, and passed further on; (6) eventually, the last member of the ring receives $v_1 \otimes \ldots v_{n-1}$; (7) the last local value is pulled; (8) it's merged into the final result v; (9) the final aggregation result tagged with version (i, j) is disseminated to everyone. Upon receiving result $(i, j), v$, a proxy places it in its queue only if it's newer than the last result $(i', j'), v'$ the proxy has ever received; this is true only if $i > i' \vee (i = i') \wedge (j > j')$.

```
01: object stable_elect(up int candidate) : s-up int leader {
02:     same int elected := 0;
03:     where (fresh elected ∧ elected ≤ candidate) // guard
04:         elected := min candidate; // aggregate
05:     leader := elected;
06: }
```

Figure 17. A simple version of the leader election protocol. Candidates with identifiers larger than the one of the elected leader (line 03) select among themselves the one that has the smallest identifier (line 04). Election result can change only when the elected leader leaves the protocol. Candidates with identifiers smaller than the leader start to compete only after all current competitors quit, causing aggregation to "reboot".

The runtime infrastructure can detect such events: it can tell if the membership dropped to zero or changed so fast that the underlying protocols couldn't propagate information across membership views. We omit the low-level details for brevity.

In the light of this discussion, β in pattern (22) is a guarded aggregation on α. Guarded aggregations are very useful; their usefulness stems from the following formal result.

THEOREM 2.1. *If flow β is a guarded, in-order aggregation over flow α using a lower bound operator $\otimes$, and α is weakly monotonic, then β is (strongly) monotonic.*

Proof. The full proof can be found in our tech report [38]. ■

COROLLARY 2.1. *In pattern* (22)*, if* α *is weakly monotonic, and* $\otimes$ *is a lower bound, then* β *is (strongly) monotonic.*

Proof. This comes from Theorem 2.1, the fact that by default aggregations are in-order, and the fact that β is guarded. ■

Now, let's analyze a few practical uses of such aggregations.

Example 3. Object $stable\_elect$ (Figure 17) implements the leader election protocol embedded in object $lock$ from Example 1. A weakly increasing input flow $candidate$ carries the identifiers of all candidates, and an increasing output flow $leader$ carries the identifier of the leader. The internal flow $elected$ is used to select the smallest identifier of a candidate (line 04); this candidate is chosen to be the leader.

Since $candidate$ is weakly increasing, we can use Corollary 2.1 to deduce that $elected$ is monotonic. Now, since aggregation is complete, once the leader starts to participate and gets elected, it bounds all further results from above with its own identifier until it quits the protocol. Hence, once the leader is elected, it continues to be elected. Candidates with identifiers smaller than the leader are held off by **where**; they wait until all others quit, causing the protocol to reboot[8]. ■

Example 4. Object $synchronize$ (Figure 18) allows a set of nodes to coordinate execution in phases, with the property that no node is permitted to enter the next phase until everyone reports they've finished working on the current one. The last phase entered by the local proxy is reported in the input flow $ready$, and the next phase the proxy is allowed to enter is reported in the output flow $phase$. The strongly increasing internal flow $done$ represents the phase all proxies entered. Monotonicity of $done$ again follows from Corollary 2.1. ■

Example 5. Object *stabilize* (Figure 19) computes the set of identifiers of packets that have been received by all nodes in the system ($stable$): it does so by intersecting sets of identifiers of packets received by individual nodes ($recv$). Flow $stable$ is strongly monotonic: packets reported as stable will forever keep being reported as such; in order words, the decision to report a packet as stable, once taken, is irreversible; this is guaranteed by Corollary 2.1. This irreversibility property is very useful: for example, if packets represent requests in a replicated database, each replica can safely process a request as soon as it knows the request is stable; it can safely do so, for other replicas will eventually also do the same. ■

Example 6. Object *decide* (Figure 20) implements a simple decision protocol. We assume that there exists a globally ordered sequence of proposals, and that nodes need to globally agree on which proposals to accept; a proposal is globally accepted only if all nodes give it a go. Global decisions are final and irreversible, and respected by everyone. Freshly joined nodes don't immediately have the rights to veto decisions, but they're always eventually recognized by the group as first-class citizens, and can henceforth veto any proposals. It is a form of consensus with dynamic membership [30].

[8] Our protocol has a weakness; it can lead to starvation. There are numerous ways to fix this; the different solutions will be discussed in a journal article.

```
01: object synchronize ( up int ready ) : s-up int phase {
02:     s-up int done = 0;
03:     where (fresh done ∧ done ≤ ready) // guard
04:         done := min ready; // aggregate
05:     phase := done + 1;
06: }
```

Figure 18. Code that coordinates processing across proxies into a sequence of synchronous phases. Phase k is completed ($done \geq k$) after all nodes report that they're ready (line 04). The system then enters the next phase (line 05). Newcomers don't get to vote on the next phase till they catch up (lice 03).

```
01: object stabilize ( up {int} recv ) : s-up {int} stable {
02:     s-up {int} recv_by_all := ∅;
03:     where (fresh recv_by_all ∧ recv_by_all ⊆ recv)
04:         recv_by_all := intersect recv;
05:     stable := recv_by_all; }
```

Figure 19. Code that determines which packets are *stable*, i.e., received by everyone in the system; this computation is an essential component of many reliable multicast protocols.

```
01: object decide ( up {int} yes, up {int} no )
02: : s-up {int} accepted, s-up {int} rejected {
03:     up {int} positive = ∅; // this proxy votes to accept
04:     up {int} negative = ∅; // this proxy votes to reject
05:     s-up {int} accept = ∅; // irreversible accept decisions
06:     s-up {int} reject = ∅; // irreversible reject decisions
07:     accepted := accept; // decided
08:     rejected := reject; // decided
09:     where (fresh accept) { // must know existing accepts
10:         positive := yes \ no ∪ accept; // catch up
11:         where (accept ⊆ positive) { // caught up
12:             accept := intersect positive; // all accepting?
13:             negative := // catch up and spread rejects
14:                 no \ accept ∪ reject ∪ union negative;
15:             where (fresh reject ∧ reject ⊆ negative)
16:                 reject := intersect negative; // all rejecting?
17: } } }
```

Figure 20. Code that makes irreversible accept/reject decisions based on local suggestions; acceptance is given only if every proxy that has synchronized with the rest gives it a go. Once a proxy is synchronized, it affects all future decisions.

The input flow yes carries sets of identifiers of proposals that the individual nodes wish to be accepted, and no carries sets of identifiers of proposals that individual nodes with to be rejected. This set encoding is similar in spirit to the one from Example 2. Output flows $accepted$ and $rejected$ carry sets of identifiers of proposals globally accepted or rejected; these are just copied from internal flows $accept$ and $reject$. Internal flows $positive$ and $negative$ carry sets of identifiers of proposals for which individual proxies will vote yes or no, respectively. They are different from yes and no because the local preferences of newly joining proxies might be ignored until they are fully synchronized with the rest of the group.

Corollary 2.1 again ensures us that $accepted$ and $rejected$ are monotonic (and that decisions are irreversible) if $positive$ and $negative$ are weakly monotonic. The latter, and the fact that $accepted$ and $rejected$ are disjoint, is a consequence of the way **where** clauses are nested, and the fact that processing each event on a proxy is done atomically. Full proof, and discussion of the forms of distributed agreement expressible in our flow language, are beyond the scope of this paper. ■

In general, monotonic aggregations based on pattern (22) and Corollary 2.1 could be used to reliably make and remember any type of consistent, irreversible distributed decisions, and retain state in the presence of churn. Other example uses include controlling atomic delivery or cleanup in a multicast protocol, total ordering, and distributed commit protocols.

Our aggregation has one powerful property we haven't revealed yet; it can be composed recursively, yielding scalable hierarchical implementations. We discuss this in Section 2.5.

2.4.4 Distributions

Whereas aggregation works to compute global values from sets of local values fed by the participants, distribution does the converse: it decomposes a global value into pieces, and passes each piece to a single participant. Flow β is a distribution of α if it can be represented as a result of the following process: we take a subset of messages $\alpha' \subseteq \alpha$, and for each message $m \in \alpha'$, we split the value $v = \nu(m)$ in this message into a set of values $v_1, \ldots v_n$ that aggregate back to v, i.e., such that $v = \otimes_{1 \le i \le n} v_i$. We then place these values in messages in β, with timestamps no smaller than $\tau(m)$.

In our language, distribution is currently only supported for set values and the union operator $\cup$; splitting a set value v means partitioning the set into subsets v_i such that $v = \cup v_i$. It is expressed by pattern **some** i (rule R09), where i is a flow name; the flow must be consistent. Each time a new set value appears in i, it is passed around, every proxy removes some elements from the set, and places them into its local queue (the process is essentially the opposite of the one Figure 16).

The space limit precludes us from discussing distribution in much detail; one example use of it is shown in Example 8.

2.5 Recursion

In Section 2.3, we explained that referring to other objects in code using pattern i_1,...i_k:=$[h]i[.i']$(x_1,...x_k) embeds proxies of object i within the proxies of the object being defined (Example 1, Figure 8). What should happen if an object recursively refers to itself? Naïve recursion, of course, should be (and is) forbidden, for it would result in infinite chains of recursively nested proxies, which are infeasible. In this section, we demonstrate that recursion can be made useful, as a way of modeling scalable, hierarchical architectures.

By default, an occurrence of i_1,...i_k:=$i[.i']$(x_1,...x_k) with object name i in a dependency, or **object** [h] i i' in the variables section, declares an embedded object that runs across the set of all recursively embedded proxies. If the declaration of an embedded object is preceded with **independently**, however, the full set of embedded proxies on different machines may be partitioned by the runtime environment into multiple

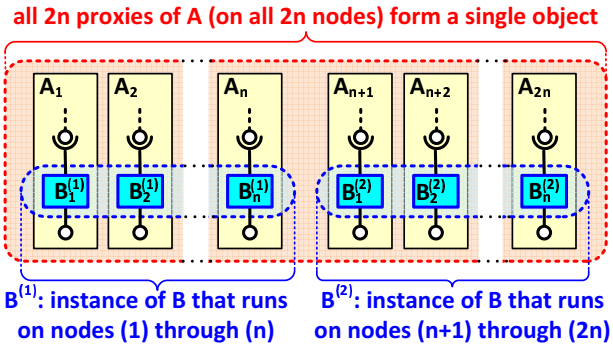

Figure 21. If the pattern i_1,...i_k:=**independently** B(x_1,...x_k) is used in the code of object A, the embedded proxies of B may be partitioned into multiple subsets (here $B_1^{(1)}, \ldots B_n^{(1)}$ and $B_1^{(2)}, \ldots B_n^{(2)}$), where each subset runs its own independent instance of object B ($B^{(1)}$ and $B^{(2)}$). The two instances don't interact with one-another; in particular, each of them performs aggregations separately, among its own proxies.

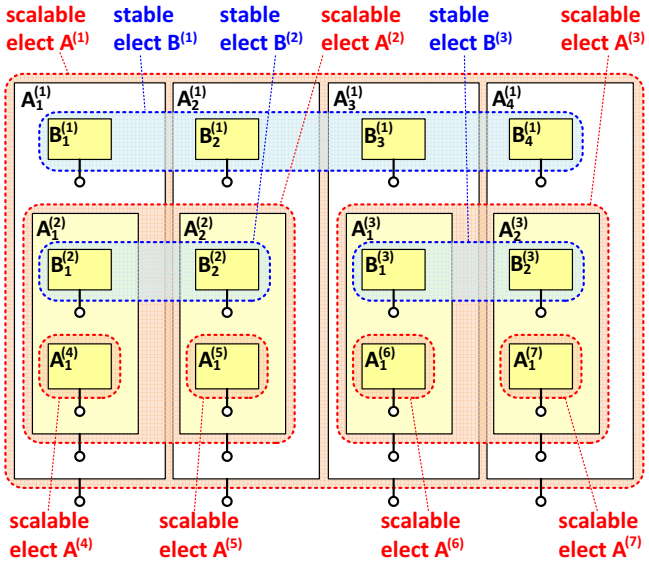

Figure 22. Object $scalable\_elect$ (Figure 23) with its recursive self-embeddings fully expanded, running on four nodes. Object $A^{(1)}$ ($scalable\_elect$) runs on all four proxies. Object $B^{(1)}$ is the instance of $stable\_elect$ embedded in $A^{(1)}$ (see line 08); it also runs on four proxies. Objects $A^{(2)}$ and $A^{(3)}$ are two instances of $scalable\_elect$ embedded in $A^{(1)}$ using **independently** (line 07); each runs on just half of the proxies. Proxies of $A^{(2)}$ do not interact with proxies of $A^{(3)}$. Objects $B^{(2)}$ and $B^{(3)}$ are the instances of $stable\_elect$ embedded in $A^{(2)}$ and $A^{(3)}$, respectively. Finally, $A^{(4)}$ and $A^{(5)}$ are the instances of $scalable\_elect$ embedded in $A^{(2)}$, whereas $A^{(6)}$ and $A^{(7)}$ are instances of $scalable\_elect$ used by $A^{(3)}$. Neither of $A^{(4)}$, $A^{(5)}$, $A^{(6)}$, or $A^{(7)}$ embeds anything; **singleton** holds on their proxies, and only the code in line 03 is active.

subgroups, each collectively running its own, independent instance of object i (Figure 21). Recursion in our language is permitted only if used with **independently**. Mutual recursion is also allowed in this case, and it's handled in the same way. Each occurrence of $i_1,...i_k:=i(x_1,...x_k)$ that would normally create a cycle must be annotated with **independently**.

If an object recursively embeds itself with this pattern, its embedded instances can also partition their sets of proxies into subgroups, and recursively embed more instances of the same object. This embedding and partitioning would continue recursively till we end up with *singleton* instances of the object, i.e., instances that run on only one proxy (Figure 22). To terminate recursion at this point, we introduce **singleton**, an expression that locally evaluates to $true$ on a proxy if it's the only one running the object instance to which it belongs.

Before we explain how we can achieve scalability through recursion, let's analyze one example use of this new feature.

Example 7. Object *scalable_elect* (Figure 23) is a hierarchical variant of the $stable\_elect$ object (Example 3). In the spirit of *divide and conquer*, object $scalable\_elect$ first partitions proxies on which it runs into subsets, and lets embedded instances of itself running on those subsets find local leaders (line 07; consult also Figure 22 and Figure 24). Each embedded instance of $scalable\_elect$ produces a consistent output flow $leader$. The internal flow *local_leader* is a union of these. Although $leader$ is strongly increasing and consistent, $local\_leader$, as a union of such flows, is not consistent, and just weakly increasing. To get a global leader, *local_leader* is fed into the original *stable_elect* (from Figure 17). ■

At first, it may not be clear that we gained much, for in the end, we still invoke $stable\_elect$, but notice that *local_leader* passed as an input to $stable\_elect$ is, in a sense, partially processed; subsets of proxies have already elected local leaders, and just a few candidates are left to compete, so $stable\_elect$ has less work to do. Indeed, for every subset of proxies created by **independently**, the output of the embedded instance of *scalable_elect* running on them is consistent: they already agreed on a leader. If so, there's no need for more than one of these proxies to feed its output to *stable_elect* (Figure 25).

If the **independently** clause never splits work among more than N instances of $scalable\_elect$ (in the example on Figure 22 we use $N = 2$), then no instance of $stable\_elect$, at any level in the hierarchy, has to run on more than N proxies (we don't need output from more than one proxy from each embedded instance of *scalable_elect*). Thus, by using recursion, we've effectively transformed a very large problem that requires a scalable protocol into a hierarchy of subproblems that can be handled by our non-scalable object $stable\_elect$.

The observation we just made can be generalized: it's not hard to see that if every input flow of an object is consistent, and the object doesn't internally use inconsistent flows such as **id**, then the output flows are also consistent, and the values in these output flows will not change if a single proxy leaves. In other words, if the object consumes only consistent flows,

```
01: object scalable_elect(int candidate) : same int leader {
02:     where (singleton)
03:         leader := candidate;
04:     elsewhere {
05:         int local_leader;
06:         local_leader :=
07:             independently scalable_elect(candidate);
08:         leader := stable_elect(local_leader);
09: } }
```

Figure 23. A hierarchical variant of leader election that uses recursion (line 07) to partition its work, and then employs the nonscalable $stable\_elect$ to combine partial results (line 08). This program can be automatically translated into hierarchical, scalable architectures similar to the one on Figure 22.

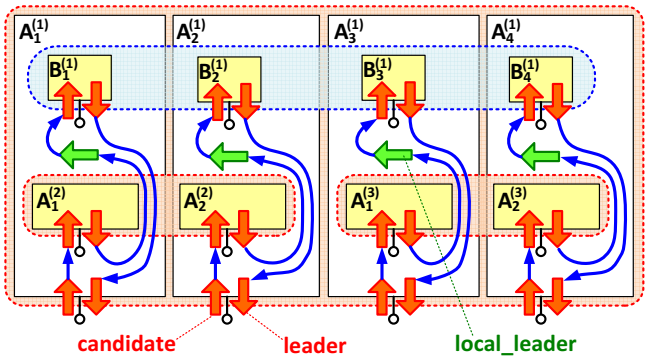

Figure 24. Dependencies in the *scalable_elect* object $A^{(1)}$, in the scenario shown on Figure 22. Candidate ids are passed to embedded *scalable_elect* objects $A^{(2)}$ and $A^{(3)}$ (line 07). The output is copied to $local\_leader$, and fed to the embedded $stable\_elect$ object $B^{(1)}$; its output is the result (line 08).

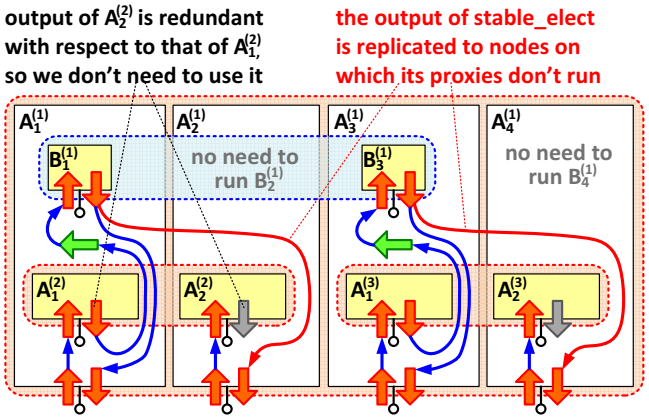

Figure 25. The flow appearing at the output of proxies $A_1^{(2)}$ and $A_2^{(2)}$ of the embedded object $A^{(2)}$ is consistent. There is no need for more than one of the proxies to feed its output to the embedded $stable\_elect$ object $B^{(1)}$. Instead, only one of the outputs is used. Proxy $B_2^{(1)}$ is never created. The output that $B_2^{(1)}$ was supposed to produce is taken from proxy $B_1^{(1)}$.

then no particular proxy has anything to offer over any of the other proxies; they're all processing the same information. In fact, it suffices if one proxy does the job and delivers the result to all the others. This is exactly what happened in our example on Figure 25: the output flow $leader$ of the embedded $scalable\_elect$ instance $A^{(2)}$ is consistent, so there is no need for both $A_1^{(2)}$ and $A_2^{(2)}$ to feed their outputs to $B^{(1)}$. The compiler and runtime can determine this fully automatically, simply by looking at the types of flows in the protocol code. Indeed, proxy elimination we just discussed is essentially a compiler optimization: we transform an executable structure (a graph of proxies, message queues, and the links between them) generated from the source code (Figure 23) so that we can improve performance without modifying the semantics.

Due to limited space and to keep our presentation simple, we omit the detailed step-by-step description of how a hierarchical structure of the sort shown here is deployed; the key ideas and the correctness argument can be found in our other work [36, 38]. Indeed, the reader will undoubtedly have noticed that there are many possible ways to build and maintain such hierarchies. In the introduction, we postulated the separation of concerns between the semantics of the protocol and implementation details such as the method of aggregation or the way nodes are organized into scalable structures. This is precisely what we have achieved by building semantics upon the abstract concept of aggregation and abstracting away hierarchy via recursion. The semantics of code on Figure 23 won't change if aggregations are performed using trees instead of token rings, if **independently** partitions a particular set of proxies into more or fewer subsets, or if the hierarchy has more or fewer layers, or if it is imbalanced. Moreover, as demonstrated through examples, the same tiny set of simple language constructs is used across a variety of protocols, and every compiler optimization or runtime mechanism we develop is going to automatically benefit all these protocols.

To conclude, we'll present one example of a hierarchical protocol that can benefit from certain types of optimizations.

Example 8. Object *scalable_repair* (Figure 26) is a hierarchical variant of the loss recovery object from Example 2. Each instance of *local_repair* computes the set of identifiers of packets that are *stable* (received everywhere) and *seen* (received somewhere) in the subset of proxies it spans; this is done by first determining which packets are stable and seen on subsets of proxies (this is done by recursively embedded instances *local_repair*, line 20), and then aggregating partial results using $\cap$ for $stable$ (a packet is stable on all proxies if it's stable on each of the subsets of proxies) and $\cup$ for $seen$ (a packet is seen among all proxies if it's seen in any of the subsets) (lines 23 and 24). Each instance of $local\_repair$ also elects one contact address ($leader$) at which one of its proxies can receive packets forwarded from elsewhere (line 22).

Similarly to instances of $scalable\_elect$ (Figure 22), here different instances of $local\_repair$ also span different portions of the system: the instance embedded directly in *scalable_repair* spans the entire system (it runs on all nodes involved in loss recovery), the instances recursively embedded span parts of it, and those within which **singleton** holds span individual nodes. The $stable$, $seen$, and $leader$ values flowing at the outputs of all the different instances of *scalable_elect* thus represent aggregate states, calculated hierarchically, bottom-up, for larger and larger portions of the system.

```
01: object scalable_repair(int addr, {int} recv)
02:     : {(int, {int})} fwd {
03:     same int leader;
04:     same {int} seen, stable;
05:     fwd, leader, seen, stable := local_repair(addr, recv, ∅);
06: }
07: object local_repair(int addr, {int} recv,
08:     same {(int, {int})} todo)
09:     : {(int, {int})} fwd, same int leader,
10:     same {int} seen, same {int} stable {
11:     where (singleton) {
12:         fwd := todo;
13:         seen := recv;
14:         stable := recv;
15:     } elsewhere {
16:         {(int, {int})} local_todo;
17:         int local_leader;
18:         {int} local_seen, local_stable;
19:         fwd, local_leader, local_seen, local_stable :=
20:             independently local_repair(
21:                 addr, recv, local_todo);
22:         leader := min local_leader;
23:         seen := union local_seen;
24:         stable := intersect local_stable;
25:         local_todo := some todo ∪ {(other local_leader,
26:             local_stable \ other local_seen) });
27: } }
```

Figure 26. A hierarchical version of the loss recovery object $repair$ from Figure 10. Object $scalable\_repair$ acts merely as a wrapper to $local\_repair$; the latter does the actual work.

In every instance of $local\_repair$, proxies compare their status (line 26). If one finds packets that are stable on its own portion of the network, but not seen in another portion of the network, it generates a forwarding request. The destination is set to be the contact address for the portion of the network that is missing packets (line 25). The request is then pushed top-down, along with some of the requests from upper levels in the hierarchy (note the use of distribution in line 25). ■

In essence, this protocol tries to recover packets as locally as possible: only if an entire group of nodes has missed the packet, the instance of $local\_repair$ running on them reports the packet as not *seen*, thus prompting some proxy elsewhere in the system to forward it. Now, suppose that **independently** partitions proxies in such a way that proxies closer in latency or some other network metric are more likely to be clustered together. Consequently, packet forwarding requested by our protocol will more likely occur between pairs of nodes close to one-another; the protocol will thus become locality-aware.

3. Related Work

Most of the existing protocol-modeling languages are based on the *finite state machine* (FSM) model: every protocol participant (a *proxy* in our terminology) is represented as a finite automaton, with transitions triggered by timeouts, the receipt of network messages, or application requests. A programmer defines states and transitions, and the compiler translates the high-level FSM specification to executable code, automating aspects such as socket operations, serialization, logging, or verification. MACE [26] and nesC [18] are prominent examples of use of this approach in the context of loosely-coupled distributed systems. Earlier systems, such as Morpheus [2], RTAG [4], Esterel [6], Prolac [27], Estelle [47], SDL [47] or LOTOS [47], targeted point-to-point protocols such as TCP.

Besides translation to code, the FSM model has also been used for program analysis: high-level protocol specifications in Promela [21] and TLA [29] can be translated to FSMs for model checking. TLA is sufficiently expressive to accurately capture strong semantics such as distributed consensus [30]. Recent SOA/WS-* standards for describing peer-to-peer interactions, particularly WSCL [5], are also founded on FSM.

Researchers argued [26] that the FSM approach is natural to work with, for the FSM code resembles well-written code in Java/C++ while being far more concise. However, systems like MACE have been used mostly for loosely-coupled systems, such as DHTs or overlays. Expressing complex DMPs such as reliable multicast or agreement via states, transitions and point-to-point messages could be quite hard [11, 20, 23]. Also, as noted earlier, code that implements core semantics (making decisions, reconfiguration, state recovery) is mixed with code that builds distributed structures for dissemination or aggregation. To achieve the sort of concern separation we advocated earlier, we need a higher-level language.

P2 [31] is a higher-level model: it replaces explicit point-to-point communication with rules in Datalog that create dependencies between local variables at different nodes (similar in spirit to our non-local dissemination and transformation); point-to-point communication is then generated automatically. This results in compact code, but operating at this level, without tools such as consistent aggregations or membership that are built into our language, it may be hard or impossible to achieve stronger semantics; indeed, P2 has been used primarily in the context of overlays, DHTs, and routing. The same issue occurs with languages based on process calculi; they cannot express strong semantics [16]. In contrast to all these approaches, our language supports consistent aggregation, recursion, batched processing (via set arithmetic), and essential object-oriented features such as encapsulation.

There's been much work on embedding group-like distributed abstractions in higher-level languages such as ML [28] or Java [14]; surveys can be found elsewhere [8, 36]. Unlike our work, these weren't designed to construct protocols, but rather to embed entire existing protocols in strongly typed or object-oriented languages. BAST [17] goes further, in that it supports typed compositions, but protocol code in BAST is written in Java, much like in other composition frameworks: Spread [3], Ensemble [20], and Appia [34]. The reasons why we prefer a dedicated language have been articulated earlier.

Our flow dependencies are functional in spirit; in this sense, our work was inspired by I/O automata (IOA) [32], which pioneered the idea of modeling entire distributed systems as components that operate on event streams. However, IOA is a specification language, and doesn't automatically yield executable code. Also, in comparison to IOA, our work is less focused on individual endpoints and their state, and more focused on flows. This creates flexibility that can be exploited to achieve the concern separation we postulated: we can run the same program over different aggregation, dissemination, batching mechanisms, or differently constructed hierarchies.

Data flows in the sense of asynchronous, massively parallel, pipelined processing, have a long tradition in areas such as VLSI or DBMS. They have also been applied to networking, e.g., in Click [35], and distributed computing, e.g., in P2 [31]. Data flows in those systems, however, are not *distributed* in the same sense as how we've defined it in Section 2.2: they are point-to-point event streams, and transformations on them are local. Although distributed data flow query engines such as Gamma [12], Volcano [19], or PIER [22] support the concept of hierarchical aggregation, they have been designed for data mining, not distributed coordination, and lack strong consistency properties of the sort discussed in Section 2.4.3. The same is true of aggregations in the context of sensor networks [10]; the properties targeted by those systems revolve around security, whereas our model is focused on reliability.

Many specific solutions employed in our work have been inspired by prior research: the use of set arithmetics in SETL [42], event-driven computing in SEDA [48], rule-based computing in Rête [15] and concern separation in aspect-oriented programming (AOP) [25], to name a few.

4. Conclusions

We proposed a new type of a programming language for distributed computing that abstracts away low-level details such as point-to-point communication, while retaining sufficient expressiveness to model complex DMPs such as distributed locking, agreement, election, or reliable multicast. Focusing on data flows, their functional dependencies, and distributed constructs such as consistent aggregation, and moving away from endpoint-centric aspects such as states and transitions, allows us to separate semantics from details such as methods of aggregation, construction or maintenance of the hierarchy. Our distributed data flow concept promotes concise code and can facilitate formal reasoning about global system behavior.

Acknowledgments

This work has been supported by grants from AFRL, AFOSR, NSF, and by the Intel corporation. We'd like to thank Lonnie Princehouse and Robbert van Renesse for their comments.

References

[1] Live Distributed Objects. *http://liveobjects.cs.cornell.edu/*.

[2] M. Abbott and L. Peterson. A language-based approach to protocol implementation. *TONS*, 1993.

[3] Y. Amir and J. Stanton. The Spread wide area group communication system. *Johns Hopkins Univ. Tech Report*, 1998.

[4] D. Anderson. Automated protocol implementation with RTAG. *TSE*, 1988.

[5] A. Banerji et al. Web Services Conversation Language. *http://www.w3.org/TR/wscl10/*.

[6] G. Berry. The foundations of Esterel. *MIT Press*, 1998.

[7] K. Birman. The process group approach to reliable distributed computing. *CACM*, 36(12):37–53, 1993.

[8] J. Briot, R. Guerraoui, and K. Lohr. Concurrency and distribution in object-oriented programming. *CSUR*, 1998.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, 2006.

[10] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. *CCS*, 2006.

[11] G. Chockler, I. Keidar, and W. Vitenberg. Group communication specifications: A comprehensive study. *CSUR*, 2001.

[12] D. DeWitt et al. Gamma - a high performance dataflow database machine. *VLDB*, 1986.

[13] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera. Enabling massively multiplayer online gaming applications on a P2P architecture. *ICIA*, 2005.

[14] P. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: abstractions for publish/subscribe interaction. *ECOOP*, 2000.

[15] C. Forgy. On the efficient implementation of production systems. *Ph.D. thesis, CMU*, 1979.

[16] R. Fuzzati and U. Nestmann. Much ado about nothing? *http://www.brics.dk/NS/05/3/*, 1995.

[17] B. Garbinato and R. Guerraoui. Using the strategy pattern to compose reliable distributed protocols. *COOTS*, 1997.

[18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *PLDI*, 2003.

[19] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD*, 1990.

[20] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. *TACAS*, 1999.

[21] G. Holzmann. The model checker spin. *TSE*, 1997.

[22] R. Huebsch et al. The architecture of pier: an internet-scale query processor. *CIDR*, 2005.

[23] D. Karr. Specification, composition, and automated verification of layered communication protocols. *Ph.D. dissertation*.

[24] A. Kay. The early history of smalltalk. *HOPL*, 1993.

[25] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. *ICSE*, 2005.

[26] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. *PLDI*, 2007.

[27] E. Kohler, F. Kaashoek, and D. Montgomery. A readable TCP in the prolac protocol language. *SIGCOMM*, 1999.

[28] C. Krumvieda. Distributed ml: Abstractions for efficient and fault-tolerant prgramming. *Cornell University Technical Report*, 1993.

[29] L. Lamport. The temporal logic of actions. *TOPLAS*, 1994.

[30] L. Lamport. The Part-Time Parliament. *TOCS*, 1998.

[31] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SOSP*, 2005.

[32] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *PODC*, 1987.

[33] S. Maffeis and D. Schmidt. Constructing reliable distributed communication systems with CORBA. *IEEE Communications Magazine*, 1997.

[34] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. *ICDCS*, 2001.

[35] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SOSP*, 1999.

[36] K. Ostrowski. *Live Distributed Objects*. Ph.D. Dissertation, Cornell University, 2008. http://hdl.handle.net/1813/10881.

[37] K. Ostrowski, K. Birman, D. Dolev, and J. Ahnn. Programming with live distributed objects. *ECOOP*, 2008.

[38] K. Ostrowski, K. Birman, D. Dolev, and C. Sakoda. Achieving reliability through distributed data flows and recursive delegation. *Cornell University Technical Report*, 2009.

[39] C. Petri. Kommunikation mit automaten. *Ph. D. Thesis. University of Bonn.*, 1962.

[40] A. Rotem-Gal-Oz. Fallacies of distributed computing explained. *http://www.rgoarchitects.com/Files/fallacies.pdf*, 2006.

[41] F. Schneider. Byzantine generals in action: implementing fail-stop processors. *TOCS*, 1984.

[42] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. Programming with sets: An introduction to setl. 1986.

[43] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *TSE*, 9(3):219–228, 1983.

[44] D. Smith, A. Kay, A. Raab, and D. Reed. Croquet: a collaboration system architecture. *C5*, 2003.

[45] J. Strohm. Managing player awareness in Darkstar. *http://www.projectdarkstar.com*, 2007.

[46] E. Tanin, A. Harwood, H. Samet, S. Nutanong, and M. Truong. A serverless 3D world. *GIS*, 2004.

[47] K. Turner. Using formal description techniques: An introduction to Estelle, LOTOS and SDL.

[48] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SOSP*, 2001.

[49] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java system. *COOTS*, 1996.