Programming with Live Distributed Objects

Krzysztof Ostrowski[†], Ken Birman[†], Danny Dolev[§], and Jong Hoon Ahnn[†]

[†]Cornell University, and [§]The Hebrew University of Jerusalem
[†]{krzys, ken, ja275}@cs.cornell.edu, [§]dolev@cs.huji.ac.il

Abstract. A component revolution is underway, bringing developers improved productivity and opportunities for code reuse. However, whereas existing tools work well for builders of desktop applications and client-server structured systems, support for other styles of distributed computing has lagged. In this paper, we propose a new programming paradigm and a platform, in which instances of distributed protocols are modeled as "live distributed objects". Live objects can represent both protocols and higher-level components. They look and feel much like ordinary objects, but can maintain shared state and synchronization across multiple machines within a network. Live objects can be composed in a type-safe manner to build sophisticated distributed applications using a simple, intuitive drag and drop interface, very often without writing any code or having to understand the intricacies of the underlying distributed algorithms.

1 Motivation

It has become common to build applications in a component-oriented manner, composing reusable building blocks by binding strongly-typed interfaces. At runtime, an underlying object-oriented managed environment, such as Java/J2EE or .NET provides further checking and support. The paradigm has numerous benefits: it promotes clean, modular architectures, facilitates extensions, enables collaborative development and code reuse, and by making contracts between components explicit and their code more isolated, reduces the risk of bugs resulting from badly documented or implicit assumptions such as cross-component behavior or side effects.

Unfortunately, distributed systems developers are only able to exploit these tools in limited ways, typically wedded to client-server programming styles. Moreover, the most widely used technologies can be awkward and inflexible. For example, a developer uses different methods to access a system depending on whether it is hosted on a single remote server [6], cloned for load-balancing on a cluster [37], or using state machine replication [52]. Yet even as the available tools have standardized on these limited options, the research community is creating a wave of powerful new technologies that includes peer-to-peer and gossip protocols, multicast with various levels of consistency, ordering and timing, Byzantine state replication, distributed hash tables, credential management services, naming services, content distribution networks, etc.

Our goal is to break through this barrier by treating protocols as components in the same sense as in .NET or COM. We propose a technology in which application components and protocols are unified within a single object-oriented paradigm. Our "live distributed objects" represent running instances of distributed protocols, but they have

types and support composition, much like "ordinary" objects. While ours is certainly not the first approach to unify distributed protocols with object-oriented environments, we innovate in ways that make the solution uniquely powerful:

- *We leverage the type system without being language-specific.* Our platform offers mechanisms such as reflection and dynamic type checking, previously seen only in systems closely tied to an underlying language, such as Smalltalk, Java, ML or IOA. In our interactive GUI, type-checking prevents users from dropping objects in inappropriately. Down the road, we'll use type checking to ensure that replicated application objects use a protocol with sufficiently strong properties.
- *It can be incrementally deployed, and supports legacy applications,* including Excel spreadsheets, Oracle databases, and web services. For example, we can import data from a database, multicast it, and export it back into a set of desktop spreadsheets.
- *Our object-oriented embedding can support any distributed protocol as a reusable component.* Existing systems are protocol-agnostic only in the limited sense that users can choose among several different protocols to implement communication. For us, protocols are objects; a small shift in perspective with broad implications.
- *The approach extends from the UI to the hardware level*, whereas prior systems focused on one class of application objects, e.g. shared data structures or UI components[1]. Jini has a vision similar to ours, but is tightly bound to the client-server paradigm, whereas our model is focused on distributed multi-party protocols.
- *We support composition of behavioral protocol types*. Prior composition toolkits either lacked types, or used a limited form of typing, where the protocol type was the type of the implementing class, and composition was achieved via inheritance.
- *Our model is replication-centric.* Although many live objects don't replicate state, the handling of replication and scalability sets our solution apart from prior ones. We're able to support various replication (multicast) models, and to express this in a type system.
- *Our system may be the first drag and drop tool for type-safe protocol composition.* Drag and drop mechanisms are easy to use and yet can support sophisticated applications. For many applications, no new code is needed at all. Prior systems (including some from which we took inspiration, such as Ensemble [33], BAST [20], x-Kernel [45], and I/O automata [36]) were programmer-intensive.

Although the current system is quite usable, live objects raise a number of questions, only some of which have been addressed. The technology requires a scalable multicast layer capable of supporting very large groups, and in which a single node can join large numbers of object-groups. In work reported elsewhere, we describe *Quicksilver*, a high-performance, scalable communication layer that achieves these goals [46,47,48]. We're also collaborating with a group at INRIA/IRISA on a gossip-based infrastructure compatible with live objects; we expect this to be useful for discovering and tracking system configuration information. Looking further out, we're extending Quicksilver to support a range of reliability models (expressed in a new protocol scripting language [47]), and are implementing a new security architecture based on reflection. We also have ideas for WAN and mobile applications, debugging, performance tuning, system management, and object state persistence. However, all of these questions lie beyond the scope of the present paper.

[1] Demos of this functionality and a prototype of our platform are available on our website [34].

2 Prior Work

While we believe our work to be innovative in the ways just described, we're not the first to integrate the object-oriented and distributed programming models.

There are many language abstractions for distributed protocols, including remote objects [17, 20], fault-tolerant objects [24], multicast objects [19], asynchronous collections [9], tuple spaces [6, 38], and replicated objects driven by multicast [25, 37] or two-phase commit [34]. None matches the requirements described above. First, these abstractions are all specialized to support specific protocols. For example, asynchronous collections cannot easily be used to express two-phase commit or leader election. Second, most lack the notion of a distributed type, and in those that do, this notion is shallow, e.g. the type of a multicast object [19] is determined by the type of transmitted events, and the type of an asynchronous collection [9] is the type of the implementing class. The former definition can't convey information about subtle behaviors of protocols such as virtual synchrony [5], while the latter severely restricts reusability. Finally, most lack support for composition.

The idea of defining object types in terms of their *behaviors* is not new [55]. CSP [24] and π-calculus [41] were some of the first protocol specifications, and these early process calculi serve as a basis for recent specification efforts, such as BPEL [3], SSDL [49], and WSCL [4]. As recently noted [19], the weakness of process calculi, and specifications based on them, is that they can't express the semantics of replication or the behavior of protocols such as consensus in a clean way. For example, while BPEL is clearly strong enough to express business processes, the language defines protocols in terms of sets of participants fixed at the outset, and can't model dynamic join or leave events. It would be very hard to express replication properties, such as "*once any group member does X, eventually all operational members do too*" [12].

On the other hand, while *state-based* approaches such as I/O automata [36], CFSM [7], interface automata [1], and others [18] are very expressive, they combine functional descriptions of protocol behaviors with the specifics of their implementations expressed through state transitions. This is useful in correctness proofs, but it may be a weakness in the context of a type system. Two protocols implemented using different data structures and states can exhibit the same external behavior, e.g. "*messages are totally ordered and delivered atomically with respect to failures*". We believe that protocols that behave equivalently should be considered to have the same distributed type; state transition representations can easily obscure such relationships [27].

Live objects support an extensible style of formal behavioral specifications for group and multicast protocols [2, 12, 22, 26]. As one composes protocols, a constructive distributed type system is obtained. The type checking mechanism is itself componentized, and can be extended by developers.

The idea of building protocols from simpler components dates to the x-Kernel [45] and to systems like Ensemble [33], which constructed replication protocols from microprotocols. Among such systems, BAST [20] is closest to ours in terms of the diversity of protocols it can express, but lacks a behavioral notion of a protocol type: protocol types in BAST are determined by the types of the implementing classes, and composition is achieved by inheritance. The creators of BAST observed that in retrospect, inheritance wasn't the right mechanism for this task. We've drawn lessons from these experiences and created a model in which inheritance isn't used at all: we treat

protocols as black boxes and connect them with typed event channels in a visual designer. Our protocol objects interact via events, much as in Smalltalk [21].

Jini [57], the widely used Java-based platform in which clients access services by dynamically loading proxy code, is highly relevant prior work. The strongest contrast is that Jini has a pervasive client-server bias, making it very hard to express object replication, particularly in applications that use strong consistency or (at the other extreme) peer-to-peer protocols.

This client-server bias is visible in many ways. First, Jini lacks a rigorous notion of a group [43], and it is hard to implement consistency across a set of group members, state replication within the group, coordination, leader-election, etc. Jini's lookup, join, and discovery specifications lack membership views (needed to assign tasks to group members) and synchronized state transfer (used to initialize new group members). Moreover, Jini doesn't guarantee consistent failure detection. Thus, while services in Jini can be grouped, the mechanism lacks expressive power to facilitate building systems that use stronger forms of replication. Additionally, abstractions such as notification and transactional protocols can't be directly modeled as objects in Jini. Finally, Jini lacks distributed types and protocol composition mechanisms.

Live objects are replication-centric, with a strong notion of protocol types and composition. This makes live objects particularly appropriate for building applications in which users collaborate, share content, or engage in other kinds of peer-to-peer behaviors, (obviously we can also support traditional non-replicated and client-server behaviors). Complex protocols can be modeled as objects, in a manner that separates behavior of the protocol from its implementation.

Many of these same issues distinguish our work from WS-* standards. Elsewhere [48], we discuss issues that arise if one tries to use WS-Notification or WS-Eventing to implement live objects. We concluded that the relevant WS-* standards are tightly bound to specific protocol implementations; as written, they cannot accommodate commercially important protocols such as peer-to-peer video streaming, BitTorrent, or Byzantine replication. We've proposed an extended WS-based eventing standard matched to the work described here, and able to overcome this problem [48].

JXTA [57] is probably the most sophisticated existing collaboration technology for peer-to-peer systems, but it doesn't support stronger replication and consistency models. While JXTA does have notions such as a group and a membership view, members can have inconsistent views. Researchers have struggled to layer reliable multicast on these mechanisms [35]. Groupware toolkits, such as Croquet [53], Groove [39], and group communication [5] toolkits all support replication, and some support strong forms of consistency. However, unlike Jini, JXTA and our work, none of these is positioned as a general-purpose interoperability platform.

3 Model

3.1 Objects and Their Interactions

A *live distributed object* (or *live object*) is an instance of a distributed protocol: programming logic executed by a set of components that may reside on different nodes and communicate by sending messages over the network. For flexibility, we won't

assume that the machines running the protocol "know" about one-another or that they share any common state. Thus, a live object could be a Byzantine fault-tolerant replicated state machine, but it could also be an entity with purely local state, one that uses gossip to share data, or an IP multicast channel.

Live objects have *behavioral types*. Suppose that object A logs messages on the nodes where it runs, using a reliable, totally ordered multicast to ensure consistency between replicas. Object B might offer the same functionality, but be implemented differently, perhaps using a gossip protocol. As long as A and B offer the same interfaces and equivalent properties (consistency, reliability, etc), we consider A and B to be implementations of the same type. The concept of behavioral equivalence is the key here; we define it more carefully in section 3.2.

When node Y executes live object X, we'll say that a *proxy* of live object X is running on Y. Thus, a live object is executed by the group of its proxies (Figure 1). A proxy is a functional part of the object running on a node. When two objects have proxies on overlapping sets of nodes, their respective proxies may interact. We can think of the live objects as interacting through their proxies.

A *reference to a live object X* is a complete set of instructions for constructing and configuring a proxy of X on a node. Thus, when node Y wants to access live object X, node Y uses a reference to X as a recipe with which it can create a new proxy for X that will run locally on Y. The proxy then executes the protocol associated with X. For example, it might seek out other proxies for X, transfer the current distributed state from them, and connect itself to a multicast channel to receive updates. Unlike proxies, which can have state, references are just passive, stateless, portable recipes.

The instructions in a reference must be complete, but need not be self-contained. Some of their parts can be stored inside online repositories, from which they need to be downloaded. These repositories are themselves live objects, referenced by the objects that use them. Thus, given a reference, a node can dereference it without prior "knowledge" of the protocol. An exception is thrown if dereferencing fails (for example, if a repository containing a part of the reference is unavailable).

We model proxies in a manner reminiscent of I/O automata. A proxy runs in a virtual context consisting of a set of *endpoints:* strongly-typed bidirectional event channels, through which the proxy can communicate with other software on the same node (Figure 1). Unlike in I/O automata, a proxy can use external resources, such as local network connections, clocks, or the CPU. These interactions are not expressed in our

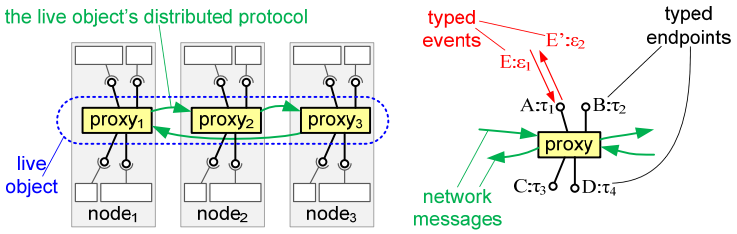

Figure 1. To access a live object (protocol), a node starts a *proxy:* a software component that runs the protocol on the node, and may communicate with proxies on other nodes by sending messages over the network. On a given node, proxies for different objects communicate via *endpoints:* strongly-typed, bidirectional event channels.

model and they are not limited in any way. However, interactions of a live object's proxy with any other component of the distributed system must be channeled through the proxy's endpoints.

All proxies of the same live object run that live object's *code*. Unlike in state machines [37, 52], we need not assume that proxies run in synchrony, in a deterministic manner, or that their internal states are identical. We do assume that each proxy of a live object X interacts with other components of the distributed system using the same set of endpoints, which must be specified as part of X's type. To avoid ambiguity, we sometimes use the term *instance of endpoint E at proxy P* to explicitly refer to a running event channel *E*, physically connected to and used by *P*.

Because our model is designed to facilitate component integration, we shall adopt a somewhat radical perspective, in which the entire system, all applications and infrastructure are composed of live objects. Accordingly, endpoints of a live object's proxy will be connected to endpoints exposed by proxies of other live objects running on the same node (Figure 2). When proxies of two different objects X and Y are connected through their endpoints on a certain node Z, we'll say that *X* and *Y* are connected on *Z*.

Example (a). Consider a distributed collaboration tool that uses reliable multicast to propagate updates between users (Figure 2). Let **a** be an application object in this system that represents a collaboratively edited document. Proxies of **a** have a graphical user interface, through which users can see the document and submit updates. Updates are disseminated to other users over a reliable multicast protocol, so that everyone can see the same contents. The system is designed in a modular way, so instead of linking the UI code with a proprietary multicast library, the document object **a** defines a typed endpoint **reliable_channel_client**, with which its proxies can submit updates to a reliable multicast protocol (event **send**) and receive updates submitted by other proxies and propagated using multicast (event **receive**). Multicasting can then be implemented by a separate object **r**, which has a matching endpoint **reliable channel**. Proxies of **a** and **r** on all nodes are connected through their matching endpoints. ■

Similarly, object **r** may be structured in a modular way: rather than being a single

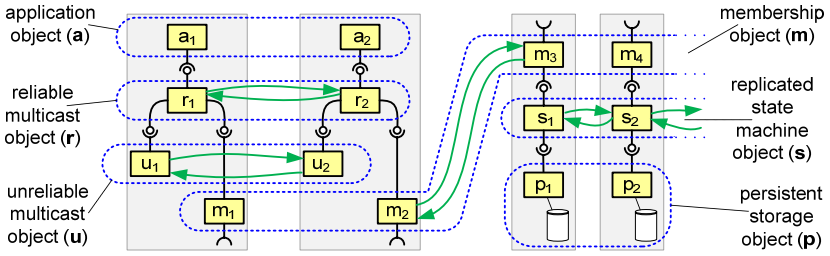

Figure 2. Applications in our model are composed of interconnected live objects. Objects are "connected" if endpoints of a pair of their proxies are connected. Connected objects can affect one-another by having their proxies exchange events through endpoints. A single object can be connected to multiple other objects. Here, a reliable multicast object **r** is simultaneously connected to an unreliable multicast object **u**, a membership object **m**, and an application object **a**. The same object can be accessed by different machines in different ways. For example, **m** is used in two contexts: by the multicast object **r**, and by replicas of a membership service. The latter employs a replicated state machine **s**, which persists its state through a storage object **p**.

monolithic protocol, **r** could internally use object **u** for dissemination and object **m** for membership tracking [12]. Additional endpoints **unreliable channel** and **membership** would serve as constracts between **r** and its internal parts **u** and **m**.

Figure 2 illustrates several features of our model. First, a pair of endpoints can be connected multiple times: there are multiple connections between different instances of the **reliable channel** endpoint of object **r** and the **reliable_channel_client** endpoint of **a,** one connection on each node where **a** runs. Since objects are distributed, so are the control and data flows that connect them. If different proxies of **r** were to interact with proxies of **a** in an uncoordinated manner, this might be an issue. To prevent this, each endpoint has a type, which constrains the patterns of events that can pass through different instances of the endpoint. These types could specify ordering, security, fault-tolerance or other properties. The live objects runtime won't permit connections between **a** and **r**, unless their endpoint types declare the needed properties.

A single object could also define multiple endpoints. One case when this occurs is when the protocol involves different roles. For example, the membership object **m** has two endpoints, for clients and for service replicas. The role of the proxy in the protocol depends on which endpoint is connected. In this sense, endpoints are like interfaces in object-oriented languages, giving access to a subset of the object's functionality. Another similarity between endpoints and interfaces is that both serve as contracts and isolate the object's implementation details from the applications using it. We also use multiple endpoints in object **r**, proxies of which require two kinds of external functionality: an unreliable multicast, and a membership service. Both are obligatory: **r** cannot be activated on a platform unless both endpoints can be connected.

Earlier, we commented that not all live objects replicate their state. We see the latter in the case of the persistent store **p.** Its proxies present the same type of endpoint to the state machine **s**, but each uses a different log file and has its own state.

Our model promotes reusability by isolating objects from other parts of the system via endpoints that represent strongly typed contracts. If an object relies upon external functionality, it defines a separate endpoint by which it gains access to that functionality, and specifies any assumptions about the entity it may be connected to by encoding them in the endpoint type. This allows substantial flexibility. For example, object **u** in our example could use IP multicast, an overlay, or BitTorrent, and as long as the endpoint that **u** exposes to **r** is the same, **r** should work correctly with all these implementations. Of course this is conditional upon the fact that the endpoint type describes all the relevant assumptions **r** makes about **u**, and that **u** does implement all of the declared properties.

3.2 Defining Distributed Types

The preceding section introduced endpoint types, as a way to define contracts between objects. We now define them formally and give examples of how typing can be used to express reliability, security, fault-tolerance, and real time properties of objects.

Formally, the type $\boldsymbol{\Theta}$ of a live object is a tuple of the form $\boldsymbol{\Theta} = (\mathbf{E}, \mathbf{C}, \mathbf{C}')$. **E** in this definition is a set of named endpoints, $\mathbf{E} = \{(\mathbf{n_1}, \boldsymbol{\tau_1}), (\mathbf{n_2}, \boldsymbol{\tau_2}), \ldots, (\mathbf{n_k}, \boldsymbol{\tau_k})\}$, where $\mathbf{n_i}$ is the name and $\boldsymbol{\tau_i}$ is the type of the i^{th} endpoint. **C** and **C′** represent sets of constraints describing security, reliability, and other characteristics of the object (**C**), and of its

environment (**C′**). **C** models constraints *provided* by the object, such as semantics of the protocol: guarantees that the object's code delivers to other objects connected to it. **C′** models constraints *required*, which are prerequisites for correct operation of the object's code. Constraints can be described in any formalism that captures aspects of object and environment behavior in terms of endpoints and event patterns. Rather than trying to invent a new, powerful formalism that subsumes all the existing ones, we build on the concepts of aspect-oriented programming [28], and we define **C** to be a finite function from some set **A** of aspects to predicates in the corresponding formalisms. For example, constraints $\mathbf{C} = \{(\mathbf{a_1}, \boldsymbol{\varphi_1}), (\mathbf{a_2}, \boldsymbol{\varphi_2}), \ldots, (\mathbf{a_m}, \boldsymbol{\varphi_m})\}$ would state that in formalism $\mathbf{a_1}$ the object's behavior satisfies formula $\boldsymbol{\varphi_1}$, and so on. We'll give examples of various practically useful formalisms and constraints later in this section.

Type $\boldsymbol{\tau}$ of an endpoint is a tuple of the form $\boldsymbol{\tau} = (\mathbf{I}, \mathbf{O}, \mathbf{C}, \mathbf{C}')$. **I** is a set of *incoming events* that a proxy owning the endpoint can receive from some other proxy, **O** is a set of *outgoing events* that the proxy can send over this endpoint, and **C** and **C′** represent constraints provided and required by this endpoint, defined similarly to constraints of the object, but expressed in terms of event patterns, not in terms of endpoints (for example, an endpoint could have an event of type *time*, and with a constraint that time advances monotonically in successive events). Each of the sets **I** and **O** is a collection of named events of the form $\mathbf{E} = \{(\mathbf{n_1}, \boldsymbol{\varepsilon_1}), (\mathbf{n_2}, \boldsymbol{\varepsilon_2}), \ldots, (\mathbf{n_k}, \boldsymbol{\varepsilon_k})\}$, where $\mathbf{n_i}$ is event name and $\boldsymbol{\varepsilon_i}$ is its type. Event types can be value types of the underlying type system, such as .NET or Java primitive types and structures, or types described by WSDL [13] etc., but not arbitrary object references or addresses in memory. We assume that events are serializable and can be transmitted across the network or process boundaries. References to live objects are also serializable, hence they can also be passed inside events. The subtyping relation on the event types is inherited from the underlying type system.

The purpose of creating endpoints is to connect them to other, matching endpoints, as described in Section 3.1 and illustrated on Figure 2. **Connect** is the only operation possible on endpoints. We say that endpoint types $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$ *match*, denoted $\boldsymbol{\tau_1} \propto \boldsymbol{\tau_2}$, when the following two conditions hold.

1. For each output event **n** of type $\boldsymbol{\varepsilon}$ of either endpoint, its counterpart must have an input event with the same name **n**, and with either type $\boldsymbol{\varepsilon}$, or some supertype of $\boldsymbol{\varepsilon}$. This guarantees that all events can be delivered between the two connected proxies.
2. The provided constraints of each of the endpoints must imply (be no weaker than) the required constraints of the other. This ensures that the endpoints mutually satisfy each other's requirements.

Formally, for $\boldsymbol{\tau_1} = (\mathbf{I_1}, \mathbf{O_1}, \mathbf{C_1}, \mathbf{C_1}')$ and $\boldsymbol{\tau_2} = (\mathbf{I_2}, \mathbf{O_2}, \mathbf{C_2}, \mathbf{C_2}')$ we define:

$$\boldsymbol{\tau_1} \propto \boldsymbol{\tau_2} \Leftrightarrow \mathbf{O_1} \rightarrow^* \mathbf{I_2} \wedge \mathbf{O_2} \rightarrow^* \mathbf{I_1} \wedge \mathbf{C_1} \Rightarrow^* \mathbf{C_2}' \wedge \mathbf{C_2} \Rightarrow^* \mathbf{C_1}'. \quad (1)$$

Relation $\rightarrow^*$ between two sets of named events expresses the fact that events from the first can be understood as events from the second. Formally, we express it as follows:

$$\mathbf{E} \rightarrow^* \mathbf{E}' \Leftrightarrow \forall\, (\mathbf{n}, \boldsymbol{\varepsilon}) \in \mathbf{E}\ \exists\, (\mathbf{n}, \boldsymbol{\varepsilon}') \in \mathbf{E}' \text{ such that } \boldsymbol{\varepsilon} \leq \boldsymbol{\varepsilon}'. \quad (2)$$

Operator "≤" on types always represents the relation of subtyping in this paper.

Relation $\Rightarrow^*$ between two sets of constraints expresses the fact that the constraints in the first set are no weaker than constraints in the second. Formally, we write this as:

$$\mathbf{C} \Rightarrow^* \mathbf{C}' \Leftrightarrow \forall\, (\mathbf{a}, \boldsymbol{\varphi}') \in \mathbf{C}'\ \exists\, (\mathbf{a}, \boldsymbol{\varphi}) \in \mathbf{C} \text{ such that } \boldsymbol{\varphi} \Rightarrow_{\mathbf{a}} \boldsymbol{\varphi}'. \quad (3)$$

Relation $\Rightarrow_{\mathbf{a}}$ is simply a logical consequence in formalism **a**. Intuitively, this definition states that if $\mathbf{C}'$ defines a constraint defined in some formalism, then **C** must define a constraint that is no weaker than that, in the same formalism. For example, if $\mathbf{C}'$ defines some reliability constraint expressed in temporal logic, then **C** must define an equivalent or stronger constraint, also in temporal logic, in order for $\mathbf{C} \Rightarrow^* \mathbf{C}'$ to hold.

For a pair of endpoint types $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$, the former is a subtype of the latter if it can be used in any context in which the latter can be used. Since the only possible operation on an endpoint is connecting it to another, matching one, hence $\boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$ holds iff $\boldsymbol{\tau_1}$ matches every endpoint that $\boldsymbol{\tau_2}$ matches, i.e. $\boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$ iff $\forall_{\tau'}\ (\boldsymbol{\tau_2} \propto \boldsymbol{\tau'}) \Rightarrow (\boldsymbol{\tau_1} \propto \boldsymbol{\tau'})$, which after expanding the definition of "$\propto$" can be formally expressed as follows:

$$\boldsymbol{\tau_1} \leq \boldsymbol{\tau_2} \Leftrightarrow \mathbf{O_1} \rightarrow^* \mathbf{O_2} \wedge \mathbf{I_2} \rightarrow^* \mathbf{I_1} \wedge \mathbf{C_1} \Rightarrow^* \mathbf{C_2} \wedge \mathbf{C_2}' \Rightarrow^* \mathbf{C_1}'. \tag{4}$$

Intuitively, $\boldsymbol{\tau_1} \leq \boldsymbol{\tau_2}$ if (a) $\boldsymbol{\tau_1}$ defines no more output events and no fewer input events than $\boldsymbol{\tau_2}$, (b) the types of output events of $\boldsymbol{\tau_1}$ are subtypes and the types of input events of $\boldsymbol{\tau_1}$ are supertypes of the corresponding events of $\boldsymbol{\tau_2}$, and (c) the provided constraints of $\boldsymbol{\tau_1}$ are no weaker and the required constraints of $\boldsymbol{\tau_1}$ are no stronger than those of $\boldsymbol{\tau_2}$.

Subtyping for live object types is defined in a similar manner. Type $\boldsymbol{\Theta_1}$ is a subtype of $\boldsymbol{\Theta_2}$, denoted $\boldsymbol{\Theta_1} \leq \boldsymbol{\Theta_2}$, when $\boldsymbol{\Theta_1}$ can replace $\boldsymbol{\Theta_2}$. Since the only thing that one can do with a live object is connect it to another object through its endpoints, this boils down to whether $\boldsymbol{\Theta_1}$ defines all the endpoints that $\boldsymbol{\Theta_2}$ defines, and whether the types of these endpoints are no less specific, and whether $\boldsymbol{\Theta_1}$ guarantees no less and expects no more than $\boldsymbol{\Theta_2}$. Formally, for two types $\boldsymbol{\Theta_1} = (\mathbf{E_1}, \mathbf{C_1}, \mathbf{C_1}')$ and $\boldsymbol{\Theta_2} = (\mathbf{E_2}, \mathbf{C_2}, \mathbf{C_2}')$, we define:

$$\boldsymbol{\Theta_1} \leq \boldsymbol{\Theta_2} \Leftrightarrow \mathbf{E_1} \leq^* \mathbf{E_2} \wedge \mathbf{C_1} \Rightarrow^* \mathbf{C_2} \wedge \mathbf{C_2}' \Rightarrow^* \mathbf{C_1}'. \tag{5}$$

Relation $\leq^*$ between sets of named endpoints used above is defined as follows:

$$\mathbf{E} \leq^* \mathbf{E}' \Leftrightarrow \forall\ (\mathbf{n}, \boldsymbol{\tau}') \in \mathbf{E}'\ \exists\ (\mathbf{n}, \boldsymbol{\tau}) \in \mathbf{E} \text{ such that } \boldsymbol{\tau} \leq \boldsymbol{\tau}'. \tag{6}$$

The use of types in our platform is limited to checking whether the declared object contracts are compatible, to ensure that the use of objects corresponds to the developer's intentions. The live objects platform performs the following checks at runtime:

1. When a reference to an object of type $\boldsymbol{\Theta}$ is passed as a value of a parameter that is expected to be a reference to an object of type $\boldsymbol{\Theta}'$, the platform verifies that $\boldsymbol{\Theta} \leq \boldsymbol{\Theta}'$.
2. When an endpoint of type $\boldsymbol{\tau}$ is to be connected to an endpoint of type $\boldsymbol{\tau}'$, either programmatically or during the construction of composite objects described in Section 4.2, the platform verifies that the two endpoints are compatible i.e. that $\boldsymbol{\tau} \propto \boldsymbol{\tau}'$.

We believe that in practice, this limited form of type safety is sufficient for most uses. For provable security, the runtime could be made to verify that live object's code implements the declared type prior to execution. Techniques such as proof-carrying code [44] and domain-specific languages with limited expressive power could facilitate this.

3.3 Constraint Formalisms

We conclude this section with a discussion of different formalisms that can be used to express the constraints in the definition of objects and endpoints. The issue is subtle because on the one hand, a type system won't be very helpful if it has nothing to check, but on the other hand, there are a great variety of ways to specify protocol

properties. It isn't much of an exaggeration to suggest that every protocol of interest brings its own descriptive formalism to the table! As noted earlier, many prior systems have effectively selected a single formalism, perhaps by defining types through inheritance. Yet when we consider protocols that might include time-critical multicast, IPTV, atomic broadcast, Byzantine agreement, transactions, secure key replication, and many others, it becomes clear that no existing formalism could possibly cover the full range of options.

A further issue is the incompleteness of many specifications, in a purely formal sense. For example, one popular formalism is temporal logic [22,12]. Here, we assume a global time and a set of locations, and a function that maps from time to events that occur at those locations. In the context of endpoint constraints, we can think of instances of the endpoint as locations, and the endpoint's incoming and outgoing events, and explicit connect/disconnect events, as the events of the temporal logic. Constraints would be expressed as formulas over these events, identifying the legal event sequences within the (infinite) set of possible system histories.

Example (b). Consider the **reliable channel** endpoint, exposed by the reliable channel **r** in the example in Section 3.1. The endpoint's type might define one incoming event **send**(**m**) and one outgoing event **receive**(**m**), parameterized by message body **m**. Constraints provided by the channel object **r** might include a temporal logic formula stating that if event **receive**(**m**) is delivered by **r** through some of the instances of the endpoint sooner than **receive**(**m′**), then for any other instance of the endpoint, if both events are delivered, they are delivered in the same sequence. ■

Example (b) illustrates a safety property of a type for which temporal logic is especially convenient. Chockler et. al. use temporal logic to specify a range of reliable multicast protocols in [12]. However, the FLP impossibility result establishes that these protocols cannot guarantee liveness in traditional networks. Thus, while we can express a liveness constraint in such a logic, no protocol could achieve it – in effect, such a protocol type would be useless in real systems!

Temporal logic is just one of many useful formalisms. In our work on a security architecture, still underway, we're looking into using a variant of the BAN logic [9] to define security properties provided by live objects or expected from their environment. Real-time and performance guarantees are conveniently expressed as probabilistic guarantees on event occurrences, e.g. in terms of predicates such as "*at least* **p** % *of the time*, **receive**(**m**) *occurs at all endpoint instances at most* **t** *seconds following* **send**(**m**)," or "*at least* **p** % *of the time*, **receive**(**m**) *occurs at all different endpoint instances in a time window of at most* **t** *seconds*".

Yet another useful formalism would be a version of temporal logic that talks about the number of instances of different endpoints in time. For example, constraints of the sort "*at most one instance of the* **publisher** *endpoint may be connected at any given time*" could describe single-writer semantics or similar assumptions made by the protocol designer. Constraints of this sort could also express fault-tolerance properties, e.g. define the minimum number of proxies to maintain a certain replication level etc.

In general, with formalisms like those listed above, type-checking might involve a theorem prover, and hence may not always be practical. In practice, however, the majority of object and endpoint types would choose from a relatively small set of standard constraints, such as best-effort, virtually-synchronous, transactional, or atomic dissemination, total ordering of events etc. Predicates that represent common con-

straints could be indexed, and stored as macros in a standard library of such predicates, and the object and endpoint types would simply list such macros. The runtime would perform type-checking by comparing such lists, and using cached known facts, such as that a virtually synchronous channel is also best-effort reliable etc. By taking advantage of late binding and reflection, features of .NET and of most Java platforms, it is easy to make these mechanisms extensible in a "plug and play" manner. This will allow developers to introduce additional formalisms down the road.

3.4 Group Types

Readers familiar with group communication [5,11] may be concerned that although our model is fundamentally about creating and working with groups of entities (live object proxies), the type system itself lacks a rigorous notion of a group. This actually makes our model simpler and more generic, without preventing us from expressing group properties. For example, to model a virtually synchronous group, we can define a pair of endpoints **channel** and **membership**, and specify constraints on the occurrences of events on the two endpoints, as in group communication specifications [12]. Within groups of endpoints, one can use temporal logic formulas with operators such as *everywhere* and *everywhere within a membership view*, much as in [2,12,22]. To bind to such a group an object would define two matching endpoints. This approach has the advantage of generality: we can potentially express a range of group semantics.

4 Language Embeddings and Support for Composition

4.1 Language Embeddings

Our model has a good fit with modern object-oriented programming languages. There are two aspects of this embedding. On one hand, live object code can be written in a language like Java and C# (we will demonstrate this in Section 4.2). On the other hand, live objects, proxies, endpoints, and connections between them are first-class entities that can be used within C# or Java code. Their distributed types build upon and extend the set of non-distributed types in the underlying managed environment. In this section, we'll discuss each of the new programming language entities we introduce: *references to live objects*, *references to proxies*, *references to endpoint instances*, and *references to connections between endpoints*. An example of their use is shown in Code 1. We will conclude this section with a discussion of two more advanced mechanisms, *template object references* and *casting operator extensions*.

A. References to Live Objects. Operations that can be performed on these references include reflection (inspecting the referenced object's type), casting, and dereferencing (the example uses are shown in Code 1, in lines 03, 05, and 06 accordingly). Dereferencing results in the local runtime launching a new proxy of the referenced object (recall from Section 3.1 that references include complete instructions for how to do this). The proxy starts executing immediately, but its endpoints are disconnected A reference to the new proxy is returned to the caller (in our example it is assigned to a local variable **folder**). This reference controls the proxy's lifetime. When it is dis-

Code 1. An example piece of code in a language similar to C#, but with a simplified syntax for legibility. Here, "ReceiveObject" is a handler of an incoming event of a live object proxy. The event is parameterized by a live object reference "ref_object". If the reference is to a shared folder, the code launches a new proxy to connect to the folder's protocol and attaches a handler to event "AddedElement" generated by this protocol, in order to monitor this folder's contents.

```
01  void ReceiveObject(ref<liveobject> ref_object) // code of an event handler
02  {
03    if (referenced_type(ref_object) is SharedFolder)
04    {
05      ref<SharedFolder> ref_folder := (ref<SharedFolder>) ref_object;
06      SharedFolder folder := dereference(ref_folder); // creates a proxy
07      external<FolderClient> folder_ep := endpoint(folder, "folder");
08      internal<FolderClient> my_ep := new_endpoint<FolderClient>();
09      my_ep.AddedElement += ...; // here's a code that registers an event handler
10      connection my_connection := connect(folder_ep, my_ep);
11      // some code to store the newly created proxy and endpoint connection references
12    }
13  }
```

carded and garbage collected, the runtime disconnects all of the proxy's endpoints and terminates it. To prevent this from happening, in our example code we must store the proxy reference before exiting (we would do so in line 11).

Whereas a proxy must have a reference to it to remain active, a reference to a live object is just a pointer to a recipe for constructing a proxy for that object, and can be discarded at any time. An important property of object references is that they are serializable, and may be passed across the network or process boundaries between proxies of the same or even different live objects, as well as stored on in a file etc. The reference can be dereferenced anywhere in the network, always producing a functionally equivalent proxy – assuming, of course, that the node on which this occurs is capable of running the proxy. In an ideal world, the environmental constraints would permit us to determine whether a proxy actually can be instantiated in a given setting, but the world is obviously not ideal. Determining whether a live object can be dereferenced in a given setting, without actually doing so, is probably not possible.

The types of live object references are based on the types of live objects, which we will define formally below. To avoid ambiguity, if **Θ** is a live object type, and **x** is a reference to an object of type **Θ**, we will write **ref**<**Θ**> to refer to the type of entity **x**.

The semantics of casting live object references is similar to that for regular objects. Recall that if a regular reference of type **IFoo** points to an object that implement **IBar**, we can cast the reference to **IBar** even if **IFoo** is not a subtype of **IBar**, and while as a result the type of the reference will change, the actual referenced object will not. In a similar manner, casting a live object reference of type **ref**<**Θ**> to some **ref**<**Θ′**> produces a reference that has a different type, and yet dereferencing either of these references, the original one or the one obtained by casting, result in the local runtime creating the same proxy, running the same code, with the same endpoints. A reference can be cast to **ref**<**Θ**> for as long as the actual type of the live object is a subtype of **Θ**.

B. References to Proxies. The type of a proxy reference is simply the type of the object it runs, i.e. if the object is of type **Θ**, references to its proxies are of type **Θ**. Proxy references can be type cast just like live object references. One difference be-

tween the two constructs is that proxy references are local and can't be serialized, sent, or stored. Another difference is that they have the notion of a lifetime, and can be disposed or garbage collected. Discarding a proxy reference destroys the locally running proxy, as explained earlier, and is like assigning **null** to a regular object reference in a language like Java. The live object is not actually destroyed, since other proxies may still be running, but if all proxy references are discarded (and proxies destroyed), the protocol ceases to run, as if it were automatically garbage collected.

Besides disposing, the only operation that can be performed on a proxy reference is accessing the proxy endpoints for the purpose of connecting to the proxy. An example of this is seen in line 07, where we request the proxy of the shared folder object to return a reference to its local instance of the endpoint named "folder".

C. References to Endpoint Instances. There are two types of references to endpoint instances, *external* and *internal*. An external endpoint reference is obtained by enumerating endpoints of a proxy through the proxy reference, as shown in line 07. The only operation that can be performed with an external reference is to connect it to a single other, matching endpoint (line 10). After connecting successfully, the runtime returns a connection reference that controls the connection's lifetime. If this reference is disposed, the two connected endpoints are disconnected, and the proxies that own both endpoints are notified by sending explicit **disconnect** events.

An internal endpoint reference is returned when a new endpoint is programmatically created using operator **new** (line 08). This is typically done in the constructor code of a proxy. Each proxy must create an instance of each of the object's endpoints in order to be able to communicate with its environment. The proxy stores the internal references of each of its endpoints for private use, and provides external references to the external code per request, when its endpoints are being enumerated. Internal references are also created when a proxy needs to dynamically create a new endpoint, e.g. to interact with a proxy of some subordinate object that it has dynamically instantiated.

An internal reference is a subtype of an external reference. Besides connecting it to other endpoints, it also provides a "portal" through which a proxy that created it can send or receive events to other connected proxies. Sending is done simply by method calls, and receiving by registering event callbacks (line 09).

An important difference between external and internal endpoint references is that the former could be serialized, passed across the network and process boundaries, and then connected to a matching endpoint in the target location. The runtime can implement this e.g. by establishing a TCP connection to pass events back and forth between proxies communicating this way. This is possible because events are serializable.

Internal endpoint references are not serializable. This is crucial, for it provides isolation. Since any interaction between objects must pass through endpoints, and events exchanged over endpoints must be serializable, this ensures that an internal endpoint reference created by a proxy cannot be passed to other objects or even to other proxies of the same object. Only the proxy that created an endpoint has access to its portal functionality of an endpoint, and can send or receive events with it.

D. References to Connections. Connection references control the lifetime of connections. Besides disposing, the only functionality they offer is to register callbacks, to be invoked upon disconnection. These references are not strongly typed. They may be created either programmatically (as in line 10 in Code 1), or by the runtime during the construction of a composite proxy. The latter is discussed in detail in Section 4.2.

E. Template Object References. Template references are similar to generics in C# or templates in C++. Templates are parameterized descriptions of proxies; when dereferencing them, their parameters must be assigned values. Template types do not support subtyping, i.e. references of template types cannot be cast or assigned to references of other types. The only operation allowed on such references is conversion to non-template references by assigning their parameters, as described in Section 4.2.

Template object references can be parameterized by other types and by values. The types used as parameters can be object, endpoint, or event types. Values used as parameters must be of serializable types, just like events, but otherwise can be anything, including *string* and *int* values, live object references, external endpoint references etc.

Example (c). A channel object template can be parameterized by the type of messages that can be transmitted over the channel. Hence, one can e.g. define a template of a reliable multicast stream and instantiate it to a reliable multicast stream of video frames. Similarly, one can define a template dissemination protocol based on IP multicast, parameterized with the actual IP multicast address to use. A template shared folder containing live objects could be parameterized by the type of objects that can be stored in the folder and the reference to the replication object it uses internally. ■

F. Casting Operator Extensions. This is a programmable reflection mechanism. Recall that in C# and C++, one can often cast values to types they don't derive from. For example, one can assign an integer value to a floating-point type. Conversion code is then automatically generated by the runtime, and injected into this assignment. One can define custom casting operators for the runtime to use in such situations. Our model also supports this feature. If an external endpoint or an object reference is cast to a mismatching reference type, the runtime can try to generate a suitable wrapper.

Example (d). Consider an application designed to use encrypted communication. The application has a user interface object **u** exposing a **channel** endpoint, which it would like to connect to a matching endpoint of an encrypted channel object. But, suppose that the application has a reference to a channel object **c** that is not encrypted, and that exposes a **channel** endpoint of type lacking the required security constraints. When the application tries to connect the endpoints of **u** and **c**, normally the operation would fail with a type mismatch exception. However, if the **channel** endpoint of **c** can be made compatible with the endpoint of **u** by injecting encryption code into the connection, the compiler or the runtime might generate such wrapper code instead. Notice that proxies for this wrapper would run on all nodes where the channel proxy runs, and hence could implement fairly sophisticated functionality. In particular, they could implement an algorithm for secure group key replication. In effect, we are able to wrap the entire distributed object: an elegant example of the power of the model. ■

The same can be done for object references. While casting a reference, the runtime may return a description of a composite reference that consists of the old proxy code, plus the extra wrapper, to run side by side (we discuss composite references in Section 4.2). In addition to encryption or decryption, this technique could be used to automatically inject buffering code, code that translates between push and pull interface, code that persists or orders events, automatically converts event data types, and so on.

Currently, our platform uses casting only to address certain kinds of binary incompatibilities, as explained in Section 5.2. In future work, we plan to extend the platform to support more sophisticated uses of casting, e.g. as in the example above, and define rules for choosing casting operators when more that one is available.

Code 2. An example live object reference, based on a shared document template, parameterized by a reliable communication channel. The channel is composed of a dissemination object and a reliability object, connected to each other via their "UnreliableChannel" endpoints, much like **r** and **u** in Figure 2. The "ReliableChannel" endpoint of the reliability object is exposed by the channel. The dissemination object reference is to be found as an object named "MyChannel", of type "Channel", in an online repository ("Id" and "Channel" are predefined types). The reference to the repository is to be found, as an object named "QuickSilver", of type "Folder", i.e. containing channels, in another online repository, the "registry" object (see Section 4.2).

```
01  parameterized object // an object based on a parameterized template
02    using template primitive object 3 // the id of a "shared document" template
03  {
04    parameter "Channel" :
05      composite object // a complex object built from multiple component objects
06      {
07        component "DisseminationObject" :
08          external object "MyChannel" as Channel
09            from external object "QuickSilver" as Folder<Id, Channel>
10              from primitive object 2  // the id of a predefined "registry" object
11        component "ReliabilityObject" :
12          ... // specification of some loss recovery object, omitted for brevity
13        connection // an internal connection between a pair of component endpoints
14          endpoint "UnreliableChannel" of "DisseminationObject"
15          endpoint "UnreliableChannel" of "ReliabilityObject"
16        export // endpoints of the components to be exposed by the composite object
17          endpoint "ReliableChannel"   of "ReliabilityObject"
18      }
19  }
```

4.2 Construction and Composition

As noted in Section 4.1, a live object *exists* if references to it exist, and it *runs* if any proxies constructed from these references are active. Creating new objects thus boils down to creating references, which are then passed around and dereferenced to create running applications. Object references are hierarchical: references to complex objects are constructed from references to simpler objects, plus logic to "glue" them together. The construction can use four patterns, for constructing *composite*, *external*, *parameterized*, and *primitive* objects. We shall now discuss these, illustrating them with an example object reference that uses each of these patterns, shown in Code 2.

A. Composite References. A composite object consists of multiple internal objects, running side by side. When such an object is instantiated, the proxies of the internal objects run on the same nodes (like objects **r** and **u** in Figure 2). A composite proxy thus consists of multiple embedded proxies, one for each of the internal objects. A composite reference contains embedded references for each of the internal proxies, plus the logic that glues them together. In the example reference shown in lines 05 to 18 in Code 2, there is a separate section "**component** *name* : *reference*" for each of the embedded objects, specifying its internal name and reference. This is followed by a section of the form "**connection** *endpoint1 endpoint2*", for each internal connection.

Finally, for every endpoint of some embedded internal object that is to be exposed by the composite object as its own, there is a separate section "**export** *endpoint*".

When a proxy is constructed from a composite reference, the references to any internal proxies and connections are kept by the composite proxy, and discarded when the composite proxy is disposed of (Figure 3). The lifetimes of all internal proxies are thus connected to the lifetime of the composite. Embedded objects and their proxies thus play the role analogous to member fields of a regular object.

B. External References. An external reference is one that has not been embedded and must be downloaded from somewhere. It is of the form "**external object** *name* **as** *type* **from** *reference*", where *reference* is a reference to the live object that represents some online repository containing live object references, and *name* is the name of the object, the reference to which is to be retrieved from this repository. The type **Θ** of the retrieved object is expected to be a subtype of *type*, and the type of the external reference is **ref**<*type*>. One example of such a reference is shown in lines 08 to 10, and another (embedded in the first one) in lines 09 to 10.

The repository could be any object of type **Θ** ≤ **folder**, where type **folder** is a built-in type of objects with a simple dictionary-like interface. Objects of this type have an endpoint with input event **get**(**n**) and with output events **item**(**n**, **r**) and **missing**(**n**). To retrieve an external reference, the runtime creates a repository object proxy from the embedded reference, runs it, connects to its folder endpoint, submits the **get** event, and awaits response. Once the response arrives, the repository proxy can be discarded.

The "**as** *type*" clause allows the runtime to statically determine the type of the reference without having to engage in any protocol. In case of composite, parameterized, or primitive references, the runtime can derive the type right from the description. The "**as** *type*" clause can still be used in the other categories of references as an explicit type cast, in case it is necessary e.g. to hide some of the object's endpoints.

The types in the reference (such as **Channel** in line 08 or **Folder**<**Id**, **Channel**> in line 09) could either refer to the standard, built-in types, or they could be described explicitly using a language based on the formalisms in Section 3.2. To keep our example simple, we assume that all types are built-in, and we refer to them by names.

C. Parameterized References. These references are based on template objects introduced in Section 4.1. They include a section "**using template** *reference*", where *reference* is an embedded template object reference, and a list of assignments to parameter values, each in a separate section of the form "**parameter** *name* : *argument*", where the *argument* could be a type description or a primitive value, e.g. an embedded

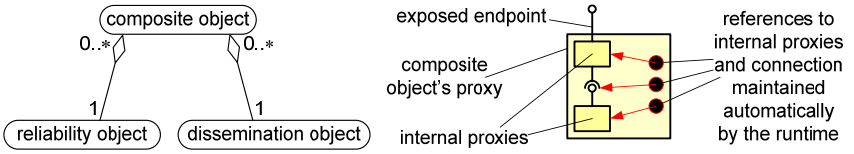

Figure 3. A live object class diagram for the composite object in Code 2 (left) and the structure of the composite proxy (right). When constructing a composite proxy, the runtime automatically constructs all the internal proxies and the internal connections between them, and stores their references in the composite proxy. Embedded proxies and connections are destroyed together with the composite proxy. The latter can expose some of the internal endpoints as its own.

Code 3. An example live object reference for a custom protocol, implemented in a library that can be downloaded from http://www.mystuff.com/mylibrary.dll. Objects running this protocol are of type "MyType1", and can be found in the library under name "MyProtocol1". The library template provides the folder abstraction introduced in Section 4.2.

```
01  external object "MyProtocol1" as MyType1 // my own, custom implementation
02    from parameterized object // an instance of the library template
03      using template primitive object 1 // an id of a built-in library template
04    {
05      parameter "URL" : http://www.mystuff.com/mylibrary.dll
06    }
```

object reference. For example, the reference in Code 2 is parameterized with a single parameter, **Channel**. The type of the parameter needn't be explicitly specified, for it is determined by the template. In our example, the template expects a live object reference to a reliable communication channel. The specific reference used here to instantiate this template is the composite reference in lines 05 to 18.

D. Primitive References. The types of references mentioned so far provide a means for recursively constructing complex objects from simple ones, but the recursion needs to terminate somewhere. Hence, the runtime provides a certain number of built-in protocols that can be selected by a known 128-bit identifier (lines 02 and 10 in Code 2). Of course even a 128-bit namespace is finite, and many implementations of the live objects runtime could exist, each offering different built-in protocols. To avoid chaos, we reserve primitive references only for objects that either cannot be referenced using other methods, or where doing so would be too inefficient. We will discuss two such objects: the *library* template and the *registry* object.

D.1 Library. A library is an object of type **folder**, representing a binary containing executable code, from which one can retrieve references to live objects implemented by the binary. The library template is parameterized by URL of the location where the binary is located (see Code 3, lines 02 to 06). The binary can be in any of the known formats that allow the runtime to locate proxy code, object and type definitions in it, either via reflection, or by using an attached manifest (we show one example of this in Section 5.2). After a proxy of a library is created, the proxy downloads the binary and loads it. When an object reference retrieved from a library is dereferenced, the library

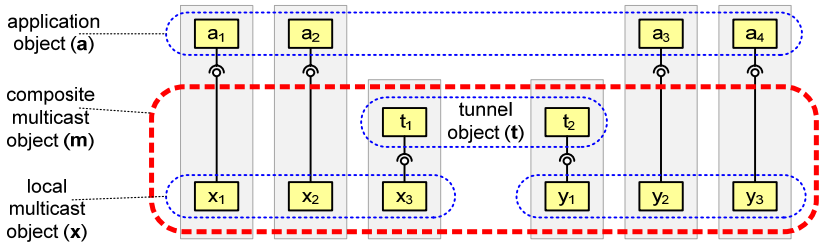

Figure 4. An example of a hybrid multicast object m, constructed from two local protocols x, y that disseminate data in two different regions of the network, e.g. two LANs, combined using a tunnel object t that acts as a repeater and replicates messages between the two LANs. Different proxies of the composite object **m**, running on different nodes, are configured differently, e.g. some use an embedded proxy of object **x**, while others use an embedded proxy of object **y**.

Code 4. A portable reference to the "hybrid" object **m** from Figure 4 built using the registry.

```
01  external object "MyChannel" as Channel
02    from external object "MyPlatform" as Folder<Id, Channel>
03      from primitive object 2 // the registry
```

Code 5. An example of a "proper" use of the registry, to specify a locally configured multicast platform, which could then be used by external references like the one in Code 4. Here, the local instance of the communication platform is configured with the address of a node that controls a region of the Internet, from which other objects can be bootstrapped.

```
01  parameterized object
02    using template external object "MyPlatform" as Folder<Id, Channel>
03      from parameterized object // from a binary downloaded from the url below
04        using template primitive object 1 // the library template
05      { parameter "URL" : http://www.mystuff.com/mylibrary.dll }
06  { parameter "LocalController" : tcp://192.168.0.100:60000 }
```

locates the corresponding constructor in the binary, and invokes it to create the proxy.

D.2 Registry. The registry object is again a live object of type **folder**, i.e. a mapping of names to object references. The registry references are stored locally on each node, can be edited by the user, and in general, the mapping on each node may be different. Proxies respond to requests by returning the locally stored references.

The registry enables construction of complex heterogeneous objects that can use different internal objects in different parts of the network, as follows

Example (e). Consider a multicast protocol constructed in the following manner: there are two LANs, each running a local IP multicast based protocol to locally disseminate messages: local multicast objects **x** and **y** (Figure 4). A pair of dedicated machines on these LANs also run proxies of a tunneling object **t**, connected to proxies of **x** and **y**. Object **t** acts as a "repeater", i.e. it copies messages between **x** and **y**, so that proxies running both of these protocols receive the same messages. Now, consider an application object **a**, deployed on nodes in both LANs, and having some of its proxies connected to **x**, and some to **y**. From the point of view of object a, the entire infrastructure consisting of **x**, **y**, and **t** could be thought of as a single, composite multicast object **m**. Object **m** is heterogeneous in the sense that its proxies on different machines have a different internal structure: some have an embedded object **x** and some are using **y**. Logically, however, **m** is a single protocol and we'd like to be able to fully express it in our model. The problem stems from the fact that on one hand, references to **m** must be complete descriptions of the protocol, so they should have references to **x** and **y** embedded, yet on the other hand, references containing local configuration details are not portable. The registry object solves this problem by introducing a level of indirection (Code 4). ■

The reader might be concerned that the portability of live objects is threatened by use of the registry. References that involve registry now rely on all nodes having properly configured registry entries. For this reason, we use the registry sparingly, just to bootstrap the basic infrastructure. Objects placed in the registry would represent the entire products, e.g. "the communication infrastructure developed by company XYZ", and would expose the **folder** abstraction introduce earlier, whereby specific infrastructure objects can be loaded. An example of such proper use is shown in Code 5.

5 System

5.1 Embedding Live Objects into the Operating System via Drag and Drop

Our implementation of the live object runtime runs on Microsoft Windows [2] with .NET Framework 2.0. The system has two major components: an embedding of live objects into Windows drag and drop technologies, discussed here, and embedding of the new language constructs into .NET, discussed in Section 5.2.

Our drag and drop embedding is visually similar to Croquet [53] and Kansas [54], and mimics that employed in Windows Forms, tools such as Visual Studio (or similar ones for Java), and in the Object Linking and Embedding (OLE) [8], XAML [40], and ActiveX standards used in Microsoft Windows to support creation of compound documents with embedded images, spreadsheets, drawings etc. The primary goal is to enable non-programmers to create live collaborative applications, live documents, and business applications that have complex, hierarchical structures and non-trivial internal logic, just by dragging visual components and content created by others from toolbars, folders, and other documents, into new documents or design sheets.

Our hope is that a developer who understands how to create a web page, and understands how to use databases and spreadsheets as part of their professional activities, would use live objects to glue together these kinds of components, sensors capturing real-world data, and other kinds of information to create content-rich applications, which can then be shared by emailing them to friends, placing them in a shared repository, or embedding them into standard productivity applications.

Live object references are much like other kinds of visual components that can be dragged and dropped. References are serialized into XML, and stored in files with a ".liveobject" extension. These ".liveobject" files can easily be moved about. Thus, when we talk about emailing a live application, one can understand this to involve embedding a serialized object reference into an HTML email. On arrival the object can be activated in place. This involves deserializing the reference (potentially running online repository protocols to retrieve some of its parts), followed by analysis of the object's type. Live objects can also be used directly from the desktop browser interface. We configured the Windows shell to interpret actions such as doubleclick on ".liveobject" files by passing the XML content of the file to our subsystem, which processes it as described above. Note that although our discussion has focused on GUI objects, the system also supports services that lack user interfaces.

We have created a number of live object templates based on reliable multicast protocols, including 2-dimensional and 3-dimensional desktops, text notes, video streams, live maps, and 3-dimensional objects such as airplanes and buildings. These can be mashed up to create live applications such as the ones on our web site (Figure 5).

Although the images in Figure 5 are evocative of multi-user role-playing systems such as Second Life, Live Objects differ in important ways. In particular, live objects can run directly on the user nodes, in a peer-to-peer fashion. In contrast, systems such as Second Life are tightly coupled to the data centers on which the content resides and is updated in a centralized manner. In Second Life, the state of the system lives in that

[2] Porting our system from C#/.NET to Mono, to run under Linux, or building a Java/J2EE version of the runtime, shouldn't be a problem, but we haven't yet undertaken this task.

data center. Live objects keep state replicated among users. When a new proxy joins, it must obtain some form of a checkpoint to initialize itself, or starts in a **null** state.

As noted earlier, live objects support drag and drop. The runtime initiates a drag by creating an XML to represent the dragged object's reference, and placing it in a clipboard. When a drop occurs, the reference is passed on to the application handling the drop. The application can store it as XML, or it can deserialized it, inspect the type of the dropped object, and take the corresponding action based on that. For example, the spatial desktop on Figure 5, only supports objects with a 3-dimensional user interface. Likewise, the only types of objects that can be dropped onto airplanes are those that represent textures or streams of 3-dimensional coordinates. The decision in each case is made by the application logic of the object handling the drop.

Live objects can also be dropped into OLE-compliant containers, such as Microsoft Word documents, emails, spreadsheets, or presentations. In this case, an OLE component is inserted with an embedded XML of the dragged object's reference. When the OLE component is activated (e.g. when the user opens the document), it invokes the live objects runtime to construct a proxy, and attaches to its user interface endpoint (if there is one). This way, one can create documents and presentations, in which instead of static drawings, the embedded figures can display content powered by any type of a distributed protocol. Integration with spreadsheets and databases is also possible, although a little trickier because these need to access the data in the object, and must trigger actions when a new event occurs.

As mentioned above, one can drag live objects into other live objects. In effect, the state of one object contains a reference to some other live object. This is visible in the desktop example on Figure 5. This example illustrates yet another important feature. When one object contains a reference to another (as is the case for a desktop containing references to objects dragged onto it), it can dynamically activate it: dereference, and connect to the proxy of the stored object, and interact with the proxy. For example, the desktop object automatically activates references to all visual objects placed on it,

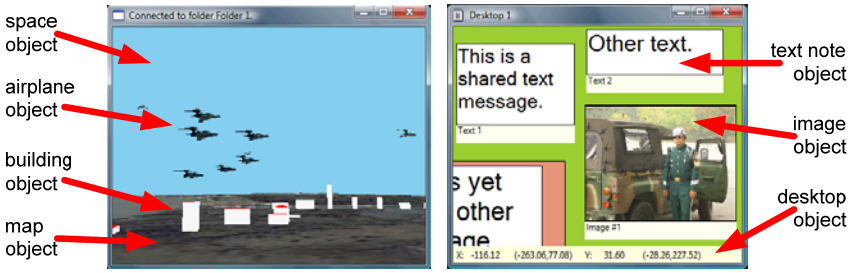

Figure 5. Screenshots of our platform running live objects with an attached user interface logic. The 3-dimensional space, the area map embedded in this space, as well as each of the airplanes and buildings (left) is a separate live object, with its own embedded multicast channel. Similarly, the green desktop, and the text notes and images embedded in it are independent live objects. Each of these objects can be viewed and accessed from anywhere on the network, and separately embedded in other objects to create various web-style mash-ups, collaborative editors, online multiplayer games, and so on. Users create these by simply dragging objects into one another. Our download site includes a short video demonstrating the ease with which applications such as these can be created.

Code 6. A .NET interface can be associated with a live object type using an "ObjectType" attribute (line 01). The interface may then be used anywhere to represent the represented live object type. The live objects runtime uses reflection to parse such annotations in binaries it loads and build a library of built-in objects, object types and templates. Object and type templates are defined by specifying and annotating generic arguments (line 03).

```
01 [ObjectTypeAttribute] // annotates "IChannel" as an alias for a live object type
02 interface IChannel<
03   [ParameterAttribute(ParameterClass.ValueClass)] MessageType>
04 {
05   [EndpointAttribute("Channel")] EndpointTypes.IDual<
06     Interfaces.IChannel<MessageType>,
07     Interfaces.IChannelClient<MessageType>>
08   ChannelEndpoint { get; } // returns an external reference to endpoint "Channel"
09 }
```

so that when the desktop is displayed, so are all objects, the references of which have been dragged onto the desktop.

By now, the reader will realize that in the proposed model, individual nodes might end up participating in large numbers of distributed protocol instances. Opening a live document of the sort shown on Figure 5 can cause the user's machine to join hundreds of instances of a reliable, totally ordered multicast protocol with state transfer, which support the objects embedded in the document. This might lead to scalability concerns. In our prior work we demonstrated that this problem is not a showstopper. Our Quicksilver Scalable Multicast (QSM) system [46], can support thousands of overlapping multicast groups, communicating at network speeds with low overhead.

5.2 Embedding Live Object Language Constructs into .NET via Reflection

Extending a platform such as .NET to support the new constructs discussed in Section 4.1 would require extending the underlying type system and runtime, thus precluding incremental deployment. Instead, we leverage the .NET reflection mechanism to implement dynamic type checking. This technique doesn't require modifications to the .NET CLR, and it can be used for other managed environments, such as Java. The key idea is to use ordinary .NET types as "aliases" representing our distributed types. Whenever such an alias type is used in a .NET code, the live objects runtime "understands" that what is "meant" by the programmer is actually the distributed type. Aliases are defined by decorating.NET types with attributes, as in Code 6 and Code 7.

Example (f). Consider a template object type **channel** for multicast channels, parameterized by the .NET type of the messages that can be transmitted. One defines an alias type as a .NET interface annotated with **ObjectTypeAttribute** (Code 6, line 01). When a library object (of Section 4.2) loads a new binary, the runtime scans the binary for .NET types annotated this way and registers them on its internal list of aliases.

Parameters of the represented live object type are modeled as generic parameters of the alias. Each generic parameter is annotated with **Parameter Attribute** (line 03), to specify the kind of parameter it represents. The classes of parameters supported by the runtime include *Value*, *ValueClass*, *ObjectClass*, *EndpointClass*, and others we won't discuss here. *Value* parameters are simply serializable values, including .NET types or

Code 7. A live object template is defined by decorating a generic class definition (line 01), its generic class parameters (line 03), and constructor parameters (line 08) with .NET attributes. To specify the template live object's type, the class must implement an interface that is annotated to represent a live object type (line 04 referencing the definition shown in Code 6). In the body of the class, we create endpoints to be exposed by the proxy (created in lines 11-12, exposed in lines 19-25), handle incoming events (line 27) and send events through its endpoints.

```
01 [ObjectAttribute("89BF6594F5884B6495F5CD78C5372FC6")]
02 sealed class MyChannel<
03   [ParameterAttribute(ParameterClass.ValueClass)] MessageType>
04 : ObjectTypes.IChannel<MessageType>, // specifies the live object type
05   Interfaces.IChannel // we implement handlers to all incoming events, see line 12
06 {
07   public MyChannel(
08     [Parameter(ParameterClass.Value)] // also a parameter of the template
09     ObjectReference<ObjectTypes.IMembership> membership_reference)
10   {
11     this.myendpoint = new Endpoints.Dual<
12       Interfaces.IChannel, Interfaces.IChannelClient>(this);
13     ... // the rest of the constructor would contain code very similar to that in Code 1
14   }
15   // this is our internal reference to the channel endpoint, the endpoint's "backdoor"
16   private Endpoints.Dual<
17     Interfaces.IChannel, Interfaces.IChannelClient> myendpoint;
18
19   EndpointTypes.IDual<
20     Interfaces.IChannel<MessageType>,
21     Interfaces.IChannelClient<MessageType>>
22   ObjectTypes.IChannel.ChannelEndpoint
23   {
24     get { return myendpoint; } // returns an external endpoint reference
25   }
26    // this is a handler for one of the incoming events of the channel endpoint
27   Interfaces.IChannel.Send(MessageType message) { ... } // details omitted
28   ... // the rest of the alias definition, containing all the other event handlers etc.
29 }
```

references to live objects, The others represent the types of values, types of live objects and types of endpoints. For example, we could define a live object type template parameterized by the type of another live object. A practical use of this is a typed folder template, i.e. a folder that contains only references to live objects of a certain type. For example, an instance of this template could be a folder that contains reliable communication channels of a particular type. Another good example is a factory object that creates references of a particular type, e.g. an object that configures new reliable multicast channels in a multicast platform.

An alias interface for a live object type is expected to specify only .NET properties, each annotated with **EndpointAttribute** (line 05). Each property defines one named endpoint for all live objects of this type. The property can only have a getter (line 08), which must return a value of a .NET type that is an alias for some endpoint type. The example in Code 6 uses alias **EndpointTypes**.**IDual**<**Interface1**, **Interface2**>. This is an alias template built into the runtime, but parameterized by two .NET interfaces.

The methods defined by these interfaces, again accordingly annotated, are used by the runtime to compile the list of this endpoint's incoming and outgoing events, and similar annotations can be used to express its constraints. When the alias defined this way is used in some context with its generic parameters assigned (lines 05-07), the runtime treats it as an alias for the specific endpoint type, with the specific events defined by those interfaces.

Having defined the object's type, we can define the object itself. This is again done via annotations. An example definition of a live object template is shown in Code 7.

A live object template is defined as a .NET class, the instances of which represent the object's proxies. The class is annotated with **ObjectAttribute** (line 01) to instruct the runtime to build a live object definition from it. This template has two parameters: the type parameter representing the type of messages carried by the channel (line 03), and a "value" parameter - the reference to the membership object that this channel should use (lines 08-09). To specify the type of the live object, line 03 inherits from an alias. This forces our class to implement properties returning the appropriate endpoints (lines 19-25). The actual endpoints are created in the constructor (lines 11-12). While creating endpoints, we connect event handlers for incoming events (hooking itself up, in line 12, and implementing these handlers, as in line 27). ■

While the use of aliases is convenient as a way of specifying distributed types, alias types are, of course, not distributed, and the .NET runtime doesn't understand subtyping rules we defined in Section 3.2. The actual type checking is done dynamically. When the programmer invokes a method of a .NET alias to request a type cast, or to create a connection between endpoints, the runtime uses its internal list of aliases to identify the distributed types involved and performs type checking by itself. The physical .NET types of aliases are irrelevant. Indeed, if the runtime determines that two different .NET types are actually aliases for the same distributed type, it will inject a wrapper code, as demonstrated below.

Example (g). Suppose that binary **Foo.dll** defines an object type alias **IChannel** as in Code example 6, and an object template alias **MyChannel** as in Code example 7. Now, suppose that a different, unrelated binary **Bar.dll** also defines an alias **IChannel** in exactly the same way, as in Code 6, and then uses this alias, e.g. in the definition of an application object that could use channels of the corresponding distributed type. If both binaries are loaded by the live objects runtime, we will end up with two distinct, binary-incompatible .NET aliases **IChannel**, representing the same distributed type. Whenever the programmer makes an assignment between these two types, the runtime dynamically creates, compiles, and injects the appropriate wrapper to forward method calls between the incompatible interfaces, to make the assignment legal in .NET. ■

6 Conclusions

Our paper described the architecture and implementation of a system supporting live distributed objects, a strongly typed, object-oriented platform in which distributed protocols are treated as first-class objects. The platform is working and quite versatile, but is still a work in progress. Future challenges include implementing our security and WAN architectures (designed but not yet operational), providing runtime monitoring and debugging tools, and automated self-configuration and tuning.

Acknowledgements. Our work was funded by AFRL/IF, AFOSR, NSF, I3P and Intel. We'd like to thank Mahesh Balakrishnan, Kathleen Fisher, Paul Francis, Lakshmi Ganesh, Rachid Guerraoui, Chi Ho, Maya Haridasan, Annie Liu, Tudor Marian, Greg Morissett, Andrew Myers, Anil Nerode, Robbert van Renesse, Yee Jiun Song, Einar Vollset, and Hakim Weatherspoon for the feedback they provided.

References

1. L. de Alfaro, T. Henzinger. Interface automata. SIGSOFT Softw. Eng. Notes 26, 5, 2001.
2. E. Anceaume, B. Charron-Bost, P. Minet, S. Toueg. On the Formal Specification of Group Membership Services. Cornell University Tech. Report TR95-1534. August, 1995.
3. T. Andrews et al. Business Process Execution Language for Web Services v1.1. May 2003. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf
4. A. Banerji et al. Web Services Conversation Language (WSCL). www.w3.org/TR/wsc110.
5. K. Birman. The Process Group Approach to Reliable Distributed Computing. Communications of the ACM 36, 12 (December 1993), pp. 37-53.
6. A. Birrell, G. Nelson, S. Owicki, W. Wobber. Network Objects. SOSP'93.
7. D. Brand, P. Zafiropulo. On communicating finite-state machines. JACM, 30(2), 1983.
8. K. Brockschmidt. Inside OLE. Microsoft Press, 1995.
9. M. Burrows, M. Abadi, R. Needham. A Logic of Authentication. TOCS 8(1):18-36, 1990.
10. N. Carriero, D. Gelernter. Linda in Context. CACM 32, 4 (Apr 1989), pp. 444-458.
11. D. Cheriton, W. Zwaenepoel. Distributed Process Groups in the V Kernel. ACM Transactions on Computer Systems 3, 2 (May 1985), pp. 77-107.
12. G. Chockler, I. Keidar, W. Vitenberg. Group Communication Specifications: A Comprehensive Study. ACM Computer Surveys, 33(4):1, pp. 43, December 2001.
13. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL). W3C Note 15 March 2001. www.w3.org/TR/wsdl
14. P. Eugster, R. Guerraoui. On Objects and Events. OOPSLA 2001, pp. 254-269.
15. P. Eugster, R. Guerraoui. Distributed Programming with Typed Events. IEEE Software, 21(2), 2004, pp. 56-64.
16. P. Eugster, H. Damm, R. Guerraoui. Towards Safe Distributed Application Development. ICSE 2004, pp. 347-356.
17. P. Eugster, R. Guerraoui, J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. ECOOP 2000, pp. 252-276.
18. X. Fu, T. Bultan, J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. WWW'03, May 20-24, 2003, Budapest, Hungary.
19. R. Fuzzati, U. Nestmann. Much Ado About Nothing? In Algebraic Process Calculi, the First Twenty Five Years and Beyond. Process algebra. www.brics.dk/NS/05/3/
20. B. Garbinato, R. Guerraoui. Using the Strategy Pattern to Compose Reliable Distributed Protocols. In Proceedings of 3rd USENIX COOTS, Portland, Oregon, June 1997.
21. A Goldberg, D Robson. Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA. 1983.
22. J. Halpern, R. Fagin, Y. Moses, M. Vardi. Reasoning about Knowledge. MIT Press, 1995.
23. J. Hickey, N. Lynch, R. van Renesse. Specifications and proofs for Ensemble layers. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99).
24. C. Hoare. Communicating sequential processes. CACM 21, 8 (Aug. 1978), p. 666-677.
25. E. Jul, H. Levy, N. Hutchinson, A. Black. Fine-Grained Mobility in the Emerald System. ACM TOCS, 6(1), pp. 109-133.
26. D. Karr. Specification, Composition, and Automated Verification of Layered Communication Protocols. Ph.D. Thesis. Cornell University. 1997.

27. I. Keidar, R. Khazan, N. Lynch, A. Shvartsman. An inheritance-based technique for building simulation proofs incrementally. ACM Trans. Soft. Eng. Methodol. 11(1):63-91, 2002.
28. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. ECOOP'97, vol.1241, pp. 220-242.
29. C. Krumvieda. Distributed ML: Abstractions for Efficient and Fault-Tolerant Prgramming. Technical Report, TR93-1376, Cornell University (1993).
30. L. Lamport. The Temporal Logic of Actions. ACM Toplas 16, 3 (May 1994), pp. 872-923.
31. B. Liskov. Distributed Programming in Argus. CACM 31, 3 (Mar. 1988), pp. 300-312.
32. B. Liskov R. Schieffler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM TOPLAS 5:3 (July 1983).
33. X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, R. Constable. Building Reliable, High-Performance Communication Systems from Components. SOSP 1999.
34. Live Objects at Cornell. http://liveobjects.cs.cornell.edu/
35. K. Loesing, G. Wirtz. An Implementation of Reliable Group Communication Based on the Peer-to-Peer Network JXTA. AICCSA 2005.
36. N. Lynch, M. Tuttle. Hierarchical correctness proofs for dist.ributed algorithms. PODC'87.
37. S. Maffeis, D. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine, vol. 14, February 1997.
38. M. Makpangou, Y. Gourhant, J.-P. Le Narzul, M. Shapiro. Fragmented Objects for Distributed Abstractions, p. 170–186. IEEE Computer Society Press, 1994.
39. Microsoft. Microsoft Office Groove. http://office.microsoft.com/en-us/groove/.
40. Microsoft. XAML Overview. http://msdn2.microsoft.com/en-us/library/ms752059.aspx
41. R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes, parts I and II. LFCS Report 89-85. University of Edinburgh, June 1989.
42. H. Miranda, A. Pinto, L. Rodrigues. Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In Proc. of 21st ICDCS, pp. 707-10, Phoenix, Arizona, 2001.
43. A. Montresor, R. Davoli, O. Babaoglu. Enhancing Jini with group communication. ICDCS Workshop, p. 69-74, April 2001.
44. G. Necula. Proof-Carrying Code. ACM SIGPLAN-SIGACT POPL '97.
45. S. O'Malley, L. Peterson. A Dynamic Network Architecture. TOCS, 10(2):110-143, 1992.
46. K. Ostrowski, K. Birman, D. Dolev. Quicksilver Scalable Multicast. *In submission.*
47. K. Ostrowski, K. Birman, D. Dolev. Declarative Reliable Multi-Party Protocols. Cornell University Technical Report, TR2007-2088. March, 2007.
48. K. Ostrowski, K. Birman, D. Dolev. Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. JWSR v. 4, no 4, Oct-Dec 2007.
49. S. Parastatidis, J. Webber, S. Woodman, D. Kuo, P. Greenfield. SOAP Service Description Language (SSDL). Technical Report, University of Newcastle, CS-TR-899, 2005.
50. M. Reiter, K. Birman. How to securely replicate services. TOPLAS 16(3): 986-1009, 1994.
51. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, D. Karr. Building Adaptive Systems Using Ensemble. Software Practice and Experience. 28(9), Aug. 1998, pp. 963-979.
52. F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. ACM Computng Surveys. 22, 4 (Dec. 1990), pp. 299-319.
53. D. Smith, A. Kay, A. Raab, D. Reed. Croquet: a collaboration system architecture. Creating, Connecting and Collaborating Through Computing, C5'2003, p. 2–9.
54. R. Smith, M. Wolczko, D. Ungar. From Kansas to Oz: Collaborative Debugging When a Shared World Breaks. CACM, April, 1997. pp 72-78.
55. A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. OOPLSA'86.
56. M. van Steen, P. Homburg, A. Tanenbaum. Globe: A Wide Area Distributed System. IEEE Concurrency, 7(1):70–78, 1999.
57. Sun Microsystems, Inc. JXTA v2.0 Protocols Specification. http://www.jxta.org
58. J. Waldo. The Jini architecture for network-centric computing. CACM 42(7):76-82, 1999.