Declarative Reliable Multi-Party Protocols

Krzysztof Ostrowski[†], Ken Birman[†], and Danny Dolev[§]
[†]Cornell University and [§]The Hebrew University of Jerusalem

Abstract

We propose a novel, declarative approach to implementing reliable multi-party protocols that enables efficient and scalable implementations. Our[1] *Properties Framework* (PF) is able to express semantics as simple as gossip or resource cleanup, or as complex as transactions, consensus, and virtual synchrony. Protocols written in the PF compile to a hierarchical, scalable runtime infrastructure. Evaluation confirms that solutions developed this way can achieve high performance, while also benefiting from better integration with the underlying runtime platform and its type system.

1 Introduction

1.1 Motivation

Building distributed systems is difficult, particularly when properties such as scalability, reliability and fault-tolerance are needed. Today, most distributed technologies are implemented from the ground up, directly over TCP and UDP (examples include group communication systems, publish-subscribe middleware and transactional systems). The resulting systems don't interoperate. We lack a solution general enough to span the full range of options, and elegantly integrated with the runtime environment so that developers can leverage available tools. The *Properties Framework* (PF) is a platform in which reliability is specified using a high-level programming language, addressing these objectives.

The work reported here was inspired by prior projects, such as MACEDON [8] and declarative networking [1] [5], which were focused on overlays. However the types of properties of interest here require forms of distributed synchronization and coordinated event handling that don't arise in overlay networks, and the PF must handle sustained, high-volume, event streams. Prior work that addressed some of these issues includes stackable micro-protocol systems, e.g. Horus [7], formal logics, e.g. TLA [4] and high-level languages, e.g. I/O automata [6], but no existing system covers the full spectrum.

[1] This research was supported by AFRL/IF with additional support from AFOSR, NSF, I3P, and Intel. Address: Department of Computer Science, Cornell University, Ithaca, NY 14850, USA; Email: krzys@cs.cornell.edu, ken@cs.cornell.edu, dolev@cs.huji.acl.il

The PF was developed with the following goals:

1. **Clarity**. The language promotes relatively simple, clear specifications even for protocols with complex synchronization requirements.
2. **Expressiveness.** The PF unifies a variety of reliability models within a single environment.
3. **Efficiency**. The automatically generated protocols can sustain high throughputs and churn.
4. **Scalability**. The PF promotes scalability in multiple dimensions.
5. **Leveraging type systems.** The PF extends a "managed" runtime environment, leveraging its type checking and debugging features.

1.2 Our Contribution

The focus of our paper is on the PF and its protocols specification language, PL. The framework combines a compiler for the language with a scalable, high-performance runtime, which is unusual because it treats protocols as a form of dataflow graph, a representation that promotes efficiency. The platform is closely integrated with an underlying type system and component composition framework: it extends the Windows .NET CLR with a new form of "distributed type", defined using PL scripts. This lets developers exploit .NET development tools, and creates new options for formal reasoning and type inference to promote robust application development.

```
protocol Cleanup {
  interface { callback Receive(int m);                (I1)
              Cleanup(int m); }                       (I2)
  properties { intset Stable, CanCleanup; }           (P1)
  bindings {
    on Receive(m) : Stable += m;                      (B1)
    on update CanCleanup(add A) :                     (B2)
      foreach (m in A) Cleanup(m); }
  rules {
    Stable := [mono,all] children(∩).Stable;          (R1)
    global.CanCleanup := Stable;                      (R2)
    CanCleanup ∪= parent.CanCleanup; } }              (R3)
```

Figure 1. A cleanup protocol. For each message m, the goal is to have components delete m only after all of them have it (liveness), while preventing any from deleting m prematurely (correctness). This logic is expressed by (R1-R3). Group membership is assumed static in this code fragment.

The PF is illustrated by the example in Figure 1, which is a fragment from a best-effort reliable multicast protocol (we'll see more of the protocol in Figure 5). The code orchestrates distributed garbage collection. An interface links to components that send and receive multicasts, which notify the code fragment through downcalls when messages are received. The PF issues an upcall when the message is globally stable and can be garbage collected.

To be effective, the PF must address a number of technical issues. The example doesn't address membership: who are the participants in the protocol? How should failures and joins (if dynamic joins are supported) be handled? We also need to understand whether this sort of protocol can scale, in several dimensions:

1. **Streams of events.** In a high performance multicast system tens of thousands of multicasts might be transmitted per second. The code fragment in Figure 1 "talks about" sets of multicasts, allowing the PF to efficiently aggregate information.
2. **Number of members.** In large groups, hierarchical structures are often needed to achieve high performance; PL code makes this natural.
3. **Reconfiguration events.** Large systems can experience high frequencies of join and leave (or failure) events. The PF has highly effective mechanisms for addressing both issues.

Other issues addressed by the PF include:

- **Conciseness**. The PL will be of little value unless complex protocols can be expressed reasonably concisely. We are finding that reliability models such as virtual synchrony, state-machine replication and transactions can be expressed with as few as 5-15 lines of declarations, plus 10-20 lines of (admittedly dense) protocol logic.
- **Heterogeneity**. Many aspects, such as the best way to deliver large amounts of data, can be customized for settings with special requirements.
- **Easy "tweaking"**. One can fine-tune a PF protocol by changing just a few lines of code.
- **Portability**. The basic runtime mechanisms to which our language compiles are very simple; they could even be offloaded to hardware.
- **Modularity**. The PF separates concerns, for example by treating data dissemination independently from coordination, promoting simplicity.

For reasons of brevity, this paper won't attempt to address every one of these issues, although we believe that the PF is successful in most of these respects (we'll mention limitations as they arise). Instead, we focus on giving the reader a strong sense of how the approach works, and we evaluate performance in some basic scenarios. Sections 2-4 describe our language, and demonstrate its simplicity and expressiveness. Section 5 outlines the underlying architecture. Section 6 discusses the scalability of our design and presents performance results.

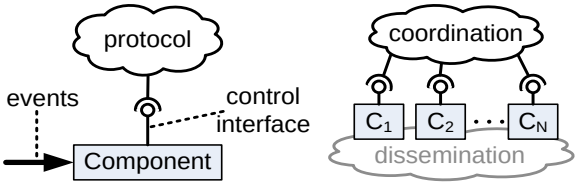

Figure 2. Components interact with protocols through control interfaces (left). Dissemination and coordination are logically independent (right).

2 System Model

2.1 Basic Definitions

We model the system as a collection of software components that reside on physical nodes and process streams of events, such as multicasts, database transactions, decisions to agree upon, requests, updates etc. A multi-party protocol is a mechanism whereby sets of components coordinate event processing.

In practice, the PF can be understood as a general-purpose engine that runs PL code on behalf of some group of components. We use the term *protocol agent* to refer to the per-protocol state and code on a component. Agents interact through a distributed data-flow architecture.

The components using our system are treated as black boxes that expose a *control interface* whereby the protocol and the component interact (Figure 2, left). A protocol can request that a component perform some action, and components can notify a protocol when events occur. For example, in the simple "cleanup" protocol (Figure 1) components report the arrival of new messages via a **Receive** event. The cleanup protocol waits until these messages become globally stable (have been received at all destinations), then signals that they can be deleted by a **Cleanup** action. The semantics can be modeled as a requirement to eventually call **Cleanup**(**m**) on each component after all have reported **Receive**(**m**) but never to invoke **Cleanup**(**m**) on any component until all have reported **Receive**(**m**).

2.2 Dissemination and Coordination

Most protocols *disseminate* events and *coordinate* the way they are processed. For example, a multicast needs to be disseminated to the receivers. The "prepare" phase of an atomic commitment protocol disseminates a question: are the participants willing to commit? The decision to garbage collect messages (because they no longer need to be available for forwarding), or to commit or abort a transaction, are examples of coordination.

The PF treats these independently (Figure 2, right). Dissemination interfaces are available with which components can send data to other components either point-to-point or as a multicast to the entire group. The default dissemination layer is *unreliable*: the PF provides tools that move data, but reliability is achieved by coordination among the group members.

Developers can extend and customize the dissemination layer. For example, a developer wishing to use BitTorrent as a dissemination technology could interface the PF to BitTorrent by adding an appropriate protocol "driver". PL protocols could then be written to create BitTorrent-based applications, but with stronger semantics than are normally available.

The default dissemination layer is optimized to move large volumes of data, taking advantage of IP multicast or overlays if appropriate. The layer will also aggregate small messages to form larger ones that make more efficient use of the available resources. Flow control is primitive in the current PF: a simple rate-control mechanism, and an interface whereby PL scripts can adjust rate parameters. But again, the modular design of the PF allows developers to change these policies if needed.

Coordination is provided by a concurrent dataflow state sharing subsystem, which is a key contribution of our approach. The basic idea is to transform each PL script into a graph that can be evaluated asynchronously. As events occur at the components of a group (the members of the multicast group in our example), they trigger chains of secondary events, much as spreadsheets update cells when something on which they depend is changed. We show that modeling synchronization protocols in this manner affords tremendous scalability opportunities.

The model is very simple and completely asynchronous: information flows up from components to a global level, then back down to the components again. Upcalls can trigger actions by components at any stage of this process. Because information moves asynchronously, in systems processing high rates of events, the PF often works with aggregated data. For example, our cleanup protocol could, at least in principle, operate on sets of messages. Notice that the code shown in Figure 1 has no direct control over when or how much aggregation occurs. Such decisions are left to the runtime platform.

2.3 Background Oracle

The basic coordination model used in the PF operates within groups of components. Membership tracking is offloaded to a fault-tolerant *membership oracle* like the one in Moshe [3] (see also [2]). Membership and configuration events such as the addition of components to the system, detection and reporting of node crashes, setting up IP multicast groups or

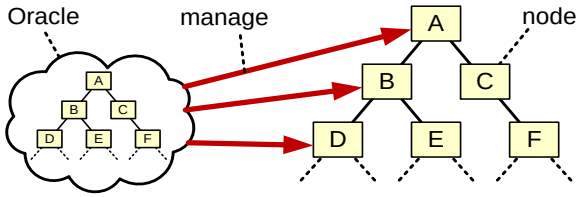

Figure 3. Tasks such as failure detection and organizing the protocol participants into distributed structures are delegated to the "Oracle:" a fault-tolerant configuration manager.

overlays, and so forth are governed by this service, which we'll call the Oracle (Figure 3). The Oracle is external to the running protocol: nodes register with it when they first launch a component that uses the PF, after which the Oracle tracks component state.

The declarative parts of a PL script "tell" the Oracle what dissemination structure is needed; the PF supports some basic options (hierarchical multicast overlays, for example). While a system is running, the Oracle receives a current event stream of configuration events from all over the system. It serializes the stream and applies these events to what can be understood as a kind of "map" of the system, showing groups of components, dissemination overlays, etc. This map advances through a sequence of revisions and as it does so, the Oracle computes required actions: group G currently consists of components A and B, but C and D should be added, and an overlay multicast tree instantiated to cover the set. It reports these to the relevant components over point-to-point channels. The associated agents then carry out their "instructions".

There isn't much magic here; oracles of this sort have been proposed in the past, although the PF takes the concept further than prior systems by treating all forms of configuration and role delegation as forms of membership events (prior membership oracles limited themselves to joins and failures).

The use of the Oracle greatly simplifies PL code. Because the Oracle informs components when the configuration changes, in a consistent and fault-tolerant manner, PL scripts don't deal with the complexities of distributed consensus. Instead, the developer starts with PL code for a "static" situation; then extends it to synchronize the protocol logic with configuration changes orchestrated by the Oracle. For example, when a component fails, we need to "terminate" the prior configuration before starting the next one, and this involves PL mechanisms for dealing with a group member that can't participate in the normal protocol because it has crashed. For a join, we face the converse problem: the new member needs to "catch up" before it can join in new protocol instances. Accordingly, one would extend the basic

code for static membership to provide handling of the exceptional conditions created by failures or joins.

To apply this approach to in our cleanup protocol, we would want to treat a failed component as if it had received any pending messages. Doing so requires a small change to the manner in which rule (R1) is coded, by adding an attribute that instructs the PF to exclude joining components from the computation. A joining member shadows protocol instances underway while it is catching up, and when it becomes synchronized with the current members, switches state, after which it will be treated as a full-fledged member. To tell the PF when this state-switch can occur one additional rule is neededd, expressing the condition for full membership. These language features[2] are also useful for handling other kinds of exceptions and configuration events.

To summarize:

1. The Oracle tells components how to configure themselves, eliminating the otherwise complex task of agreeing on the configuration. Problems such as agreement on membership, leader election and consistent configuration are automated.
2. It imposes a system-wide ordering on concurrent configuration related events such as joins, leaves, failures, and new role assignments.
3. It simplifies generation of PL code: one codes a script for a static case, and then refines it to address membership changes.

3 Our Language

3.1 Modeling Protocols through Properties

The key idea behind our proposal is the observation that the state and progress of a large class of protocols can be concisely and accurately described by sets of *properties*, and that doing enables scalable solutions in the three dimensions cited earlier. A property is just a variable managed by a component, or group of components. Properties represent two kinds of data: information obtained from a component, and information aggregated over sets of components. When a property takes on a new value, this can also represent a decision or even trigger actions at one or multiple components. This perspective lets us implement protocols as distributed computations over properties, accompanied by mechanisms that tie the values of properties to actions and events.

To see how this works, let's return to our cleanup example. Property **x.Stable** for component **x** represents the set of identifiers of messages that **x** has received. Each component **x** has its own, private instance of this property. The underlying data type is a set of identifiers, and the PL code updates **x**.**Stable** for each new message **m** received in the component (line B1). Similarly, property **CanCleanup** represents the identifiers of messages **x** is allowed to delete. Changes to **x.CanCleanup** eventually trigger appropriate actions: if **m** is added to **x.CanCleanup**, **x** eventually deletes its copy of message **m** (line B2). The cleanup semantics are now expressed as follows.

1. After **m** ∈ **x.Stable** holds for all **x**, eventually we want **m** ∈ **x.CanCleanup** to also hold for all **x**.
2. We do not want **m** ∈ **x.CanCleanup** to hold for any **x** until **m** ∈ **x.Stable** holds for all **x**.

Our basic idea is to let the protocol designer express protocols by writing down these kinds of relationships, against the backdrop of an implicit hierarchical structure, and without actually indicating *when* the rules should fire. The PF compiles the script into a dataflow graph which it executes at runtime, achieving high performance by aggregating events. Because the scripts are written in such a declarative, asynchronous manner, they have a natural match with this form of asynchronous evaluation.

3.2 Expressing the Behavior of Protocols

Conceptually, we can think of a protocol that runs on a sequence of events as a sort of concurrent distributed "engine" continuously executing the following five stages:

1. **Extracting information** from components, such as message receive events in our example.
2. **Aggregating or disseminating this information**, for example to compute the system-wide received message set.
3. **Making decisions**, e.g. deciding that a message can be cleaned up. This may require a leader, since many decisions will be non-deterministic, particularly in our asynchronous model.
4. **Disseminating decisions** so that all components see a consistent outcome.
5. **Triggering actions,** such as when components delete cached messages.

In our language, all information and decisions, including aggregate information, global decisions etc. are represented as properties. Information about an individual component is represented as a property of the component. Global information and decisions are represented as properties of the entire system. Stages 1 and 5 above are realized via mechanisms that relate the values of properties associated with the individual components to actions by entire groups of components. Stages 2, 3, and 4 are realized via mechanisms that relate the values of various properties, perhaps

[2] We don't want this paper to become a "user's manual." Accordingly, we'll keep our examples accurate, but simple, and won't explain every feature in detail.

defined in different contexts, to one-another. For example, in Figure 1, rule (B1) and (B2) implement stages 1 and 5, and rules (R1), (R2) and (R3) implement stages 2, 3 and 4, respectively.

3.3 The Structure of Protocol Specifications

A PL script includes an interface specification and property declarations, followed by a list of *bindings*, *rules*, and *conditions*.

The interface is a C-style list of methods used by the protocol to interact with components. Events reported by components to the protocol are preceded with **callback**. The others are upcalls representing actions that the protocol may request of components.

Property declarations are, again, C-style, starting with the data type. Common built-in types include **int** (numbers), and **intset** (sets of numbers). These types permit very efficient representations, and suffice for the protocols we've analyzed so far, but one can certainly imagine protocols that would need other types. The set is extensible, but on the other hand, not all types permit equally efficient implementation.

Bindings are C-like code snippets that implement the mapping between the properties and the interface. The code can be triggered by an event (**on Event**), by an update to a property (**on update Property**(details), where "details" may be of the form **add** X, **assign** Y etc. and are used to access the specifics of the update), when the component is connected to the protocol (**on initialization**) etc. The body of the code may request actions, or update values of locally defined properties.

Rules are simply assignments that update values of properties based on values of other properties. The rules represent the actual logic of the protocol.

The left side of a rule specifies the *target property* (to be updated). It is followed by an *update operator*. This can be a plain assignment (:=) or a "C-style" updating assignment: +=, -=, *=, but also ∪=, ∩= etc. The right side is an expression. Expressions in rules can only build on properties, and cannot include code snippets or calls to interface methods.

3.4 The Underlying Hierarchical Structure

We've stressed that scalability is a fundamental goal of the properties framework. Hierarchy is visible through mechanisms whereby the designer expresses parent-child relationships within PL scripts, for example by associating a global "stability" measure with the set of underlying per-component "stability" properties.

Let's start with nomenclature. The PF treats any large group of components as a hierarchy of *entities*, each representing a subset of components (Figure 4), and forming "parent-child" relationships. At the root of this hierarchy, there is an abstract "**global**" entity;

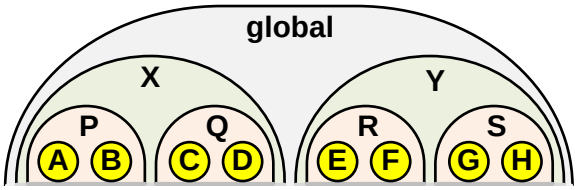

Figure 4. In our language the system is modeled as inherently hierarchical. This way, every valid protocol specification yields a hierarchical protocol.

it represents an entire component group and does not have a parent. The leaves are the individual components. In between are layers of abstract entities that represent the subsets of the component group. The Oracle constructs this hierarchy, using heuristics that try to capture "physical" structure.

Property names in rules can be qualified by keywords such as **local**, **global**, **parent**, or **children**. When we talk about a property within the children of some entity, we can also specify an aggregation operator with which the value of the property in the parent is computed from the properties of its children. In general, rules update private properties based on the values of either other local private properties, those of an entity's immediate parent (via the **parent** keyword) or those of its children (via **children**).

Properties, expressions and operators can also have *attributes*. We use a C#-like convention: attributes are surrounded with square brackets, and directly precede the object they apply to; parameters and arguments are surrounded with round brackets, and directly follow the object they apply to. For example, in (R1), the **mono** attribute applies to the property Stable, and tells the PF that Stable needs to be "monotonic"; we'll explain how and why this is done in Section 3.8.

By default, each rule listed in the protocol specification "runs" anywhere where it is well-defined. If the target property is qualified with **global**, the rule is instantiated only at the **global** entity, and if it's qualified with **local**, the rule only runs on the components. If the rule's expression uses properties qualified with the **parent** keyword, it can only be instantiated on the entities that do have a parent, and if the rule refers to properties qualified with **children**, it will only run on those entities that have children. Thus, for example, rule (R1) runs everywhere except at individual components, rule (R2) runs only in the **global** level and rule (R3) runs everywhere except at the **global** level.

3.5 How the PF Works

With this background in hand, we can get a feeling for the way that the PF operates. Assume that a group of components are launched on some set of computing nodes, and join a reliable multicast group that uses our cleanup code. Their startup triggers reg-

istration with the Oracle. As we saw previously, it tells each to instantiate a protocol agent that will run the PL script. At the same time, the Oracle organizes the components into a hierarchy of token rings, within which the PF aggregation and data-flow algorithms are implemented[3].

Now, visualize a stream of multicasts that are pouring through the group (under control of a dissemination protocol, not shown), and assume that no messages are ever lost and that no components fail or join. As messages arrive, individual components call the **Receive** interface at their local protocols agent. These interface calls are concurrent and unsynchronized. Rule (B1) fires, updating the property **Stable** at each corresponding component. **Stable** thus accumulates the set of messages that are stable at that component – that have been received locally.

Now we make use of the token rings mentioned above. The PF circulates tokens in each of these rings; they visit the protocol agents one by one, collecting the value of **Stable** at each and building an aggregate. To be stable within the group as a whole, a message must be stable at all the components. In effect, the global value of **Stable** should intersect the values of **Stable** at the full set of current members of the group. We can see the code that carries out this computation in the example. Rule (R1) tells us that **Stable** within a parent entity is the intersection of **Stable** at its children.

Recall that PL code is written as if membership does not change. Thus, the set of children can be seen as static and well defined. Our token can visit components, one by one, starting with the **Stable** property of the first component it visits, and then intersecting it with the **Stable** property at each successive component visited. When the token has visited all of some set of components, it carries a representation of the set of messages stable across them.

Where, specifically, does the hierarchy actually come from? Built into the Oracle is a mechanism tied to the way that we implemented the PF. Knowing that the PF efficiently supports trees of token rings, the Oracle structures any group of components appropriately. If the group is small enough, it will use a single level hierarchy; if not, it subdivides the group hierarchically. For large groups, the layout also mimics the physical layout: the top level might run over a WAN, while lower layers live within distinct datacenters or perhaps even on the same segment of a network. When the Oracle initializes the components, it tells each where in the hierarchy it will "live", and each component establishes a peering relationship with its neighbors in the hierarchy. Some components are assigned multiple roles: they are in leaf-level rings, but also in token rings corresponding to inner entities, and so forth. Where a property needs to be computed by a leader (for example, a decision in a non-deterministic protocol), the Oracle can assign leader status to one of these protocol agents; should it fail, the Oracle reassigns the role.

Thus, a given protocol agent plays a local role on behalf of the local component, but also can play other roles on behalf of the hierarchy as a whole. With respect to our example, tokens visit such an agent in its inner roles to aggregate Stable in these inner entities. Eventually, **global.Stable** is updated, and on this basis, **global.CanCleanup**. The tokens then carry the global information back down to the components, which triggers garbage collection.

Tokens efficiently represent aggregated information about properties: depending on the use made of a given property, it often suffices to carry just a single integer, bit-vector or other compressed representation from agent to agent. In the most complex cases, a token carries the full set of identifiers, but this is uncommon in the protocols we've analyzed to date.

3.6 The Property Aggregation Operator

The foregoing example should help build intuition. Let's look a bit more closely at the way the PF performs aggregation. First, to ensure that PL code can scale efficiently, a parent entity can never refer to a property of any of its children individually; it can only refer to an aggregate, obtained by taking the values of a certain property for the set of its children, and aggregating these values into a single value using a commutative, associative binary operator. A property qualified with **children** represents the result of such an aggregation.

For example, the value of **children(∩).Stable** on entity **x**, is the set intersection of the values of **y.Stable** for **y** iterating over the children of **x**. In the instance of rule (R1) running at the **global** context in the scenario shown on Figure 4 this expression would effectively translate to "**X.Stable ∩ Y.Stable**".

The exact way the aggregation is performed can be controlled by attributes. For example, attribute "**all**" in rule R1 on Figure 1 specifies that values from each of the child entities must be included in this aggregation (other attributes could request a quorum etc.). All is interpreted relative to the current membership of the entity, as dictated by the Oracle.

3.7 Rules as Flows of Control Information

PF rules are evaluated "continuously" but in a data-flow manner. One can think of a rule as a repre-

[3] This is probably as good a time as any to comment that PF could also do aggregation using trees or other structures. Token rings were a somewhat arbitrary decision, but have worked well for us in the examples we've explored to date.

sentation of an *information flow*. Whenever a value of any of the properties used in the rule's expression changes, the expression is eventually re-evaluated, and the value of the target property is updated. To continue our running example, assume that we are doing garbage collection in the hierarchy depicted on Figure 4. Suppose that message **m** is added to **A.Stable.** Eventually, the PF token makes a tour of the components in entity P, and by application of rule (R1), recalculates **P.Stable**. Suppose that this causes the value of **P.Stable** to change. Eventually, a token will make the rounds of components representing entity X, and now rule (R1) will be used to recalculate **X.Stable**, and so on. This way, updates of **x.Stable** at any of the components are eventually propagated by rules firing in a cascading manner, up the hierarchy. The value of **global**.**Stable** thus tracks the set intersection of **x**.**Stable** for all components **x**.

In our cleanup example, when message **m** arrives at *every* component, **global.Stable** will eventually be updated to contain **m**. This update, in turn, activates the decision rule (R2), which inserts **m** into **global.CanCleanup**. Now, the tokens carry information downward. Rule (R3) is repeatedly applied, and this finally results in **m** being added to **x.CanCleanup** for all components **x**. And when that happens, it triggers binding (B2) and the protocol invokes **Cleanup**(**m**) on each component.

3.8 Monotonicity

The term *monotonic*, applied to a property, captures the intuition that once the property holds, with respect to some object, it will continue to hold. It should be evident that some properties are inherently non-monotonic. For example, in a protocol that recovers missing messages, the set of messages needed at a component **x** would grow as new multicasts enter the system but fail to reach **x**, and shrink as multicasts get forwarded to **x**. But other properties, at least conceptually, must behave monotonically, as in the case of **x**.**CanCleanup**. Once components garbage collect a message, the actions can't be rolled back.

Unfortunately, there are situations in which an asynchronous, data-flow oriented rule evaluation strategy might fail to execute "forward" in time, and we see an instance of the problem when the global Stable is used to update the global CanCleanup property. By designating that an aggregate *must* be computed monotonically, the protocol designer forces the PF to ensure that *the values of properties of the child entities used in newer aggregations are as least as fresh as those used in older aggregations*, i.e. each child must provide value at least as fresh as any value it provided in the past. Thus the use of **mono** in our cleanup code: it forces the PF to use increasingly fresh data each time it updates **Stable**, which becomes monotonic, ensuring that **CanCleanup** will also be monotonic, as required for correctness.

Monotonicity is powerful, but can also be costly, and is not always necessary. For example, it's easy to see that rules (R2) and (R3) and binding (B2) will work correctly even without it. To enable efficient implementations where weak semantics is sufficient, we make costly features that involve synchronization, ordering etc. optional. Thus, we can express protocols that concurrently perform multiple activities that are as weakly synchronized as possible: just enough to guarantee correctness.

3.9 Handling Membership Changes

Failure handling is a source of complexity for classic implementations of the types of protocols targeted in our work. However, this is not the case in the PF. The Oracle simply excludes the failed node from the system, and the protocol continues in the modified configuration. Somewhat to our surprise, we've found that for most protocols coded with PL (not just simple ones, but also virtual synchrony and transactions). The rules can be written so that they will be correct if failed components are simply ignored by all members in a consistent manner. No explicit handling of crashes is necessary

Joins are trickier. In protocols that allow dynamic joins, new members need some form of catch-up period. In the PL, a joining component executes a subset of the rules, which allows it to "catch up" with the existing members, without contaminating global properties or destabilizing of the protocol by exposing it to initialized component states. The programmer explicitly specifies which properties and rules are "active" on joining members, and which can only activate on the "fully joined" members.

For example, in a Cleanup protocol, new members will generally not have copies of all messages. If we wanted to support joins, we would exclude new members from the stability detection. First, each rule would be annotated to warn the PF not to include properties from joining members when computing aggregations. We can also indicate rules that the joining member shouldn't execute until it catches up. Next, an additional rule (a "condition") is added specifying a logical test whereby the PF can decide when the joining member has caught up with the active members. For example, in the Cleanup protocol, a condition on joining would be that the new member have a local copy of any message that the parent entity has declared as stable.

When a joining component satisfies all conditions, the PF "promotes" it to "full" status in an asynchronous manner. This transition involves a careful synchronization: PF ensures that it happens atomically. "Catching up" can take a while; indeed, for a sys-

tem running at high event rates, a joining component might never get a chance to catch up (it would eventually throw an exception). We believe that this asynchronous catch-up mechanism is preferable to the forms of state transfer techniques commonly seen in group communication and similar systems, because those can be highly disruptive. Within the PF, a joining component can take its time obtaining the state from other members. By specifying conditions for a full fledged membership in a group, the PL script protects itself against violations of implicit invariants. Joining components can execute Pl scripts while catching up, but won't disrupt active members.

Some join and failure handling mechanisms require more complex forms of synchronization. However, our experience to date has been that even complicated coordination protocols, such as virtual synchrony, can still support join and failure with just a few additional rules and annotations.

4 Examples

We've done about as much as can be done with the garbage collection example. Accordingly, we'll now introduce additional examples that also illustrate other features of the language and framework. Earlier, we commented that developers will often write code for a static case, and then add fault-handling and performance optimizations. For simplicity, the code shown below is also for a static scenario; all three examples can be extended to handle crashes and joins using no more than two or three extra rules.

4.1 Lost-Message Repair

Figure 5 shows a protocol for peer-to-peer loss recovery, of the sort that one might combine with the cleanup protocol to build a simple reliable multicast protocol. Indeed, the protocol inherits several properties and rules from the cleanup protocol, via a notation similar to class inheritance in C#. The additional rules are used to detect forwarding opportunities, and delegate them to individual components.

Property **HeardOf** represents messages that been received by some of the components. Rule (R5) starts the process of gossiping this value and rules (R6) and (R7) disseminate it up and down the hierarchy.

Property **Missing** represents missing messages. There are two cases: messages missing at an individual component, defined to be those the component has heard of but that aren't locally stable (R8), and messages missing at all children in a higher-level of the hierarchy. The attribute [**delay**] is used to avoid flagging a message as missing "instantly": in an asynchronous system, a token could easily visit one component that has received a message and then a second one where the message hasn't yet arrived, and we don't want to trigger excessive forwarding.

```
protocol Forwarding : Cleanup {
  interface { Forward(int m, address x); }
  properties { intset HeardOf, Missing,
                      Cached,  Push[address];  }
  bindings { on update Push[x](assign S) :
    foreach (m in S) Forward(m, x); }
  rules {
    local HeardOf ∪= Stable;                    (R5)
    HeardOf ∪= children(∪).HeardOf;             (R6)
    HeardOf ∪= parent.HeardOf;                  (R7)
    local Missing := [delay] HeardOf \ Stable;  (R8)
    Missing := children(∩).Missing;             (R9)
    local Cached := Stable \ CanCleanup;        (R10)
    Cached := children(∪).Cached;               (R11)
    Push[x] := Cached ∩ peer(x).Missing;        (R12)
    Push[x] := [erase] parent(∪).Push[x]; } }   (R13)
```

Figure 5: Forwarding in a reliable multicast.

```
protocol CoordinatedPhases {
  interface { Phase(int k); }
  properties { int Last = 0, Next; }
  bindings {
    on update Next(assign k) : Phase(k); }
  rules {
     Last := [mono,all] children(min).Last;
     global.Next := Last + 1;
     Next [mono] := parent.Next;
     local.Last := Next; } }
```

Figure 6. Entering phases in coordinated manner.

Rules (R10-R11) are used to determine which messages are cached "somewhere" within a given entity. Finally, rules (R12-R13) arrange for a "peer" of a component missing a message to forward it.

4.2 Coordinated Phases

Many of the behaviors the PF is intended to model – virtual synchrony, state machines – involve a form of barrier synchronization behavior. Figure 6 shows a PL specification of a protocol that allows a set of processes to perform some action in "phases". The interface between the protocol and the components is a single method call, **Phase**(**int** k), where **k** is the number of the processing phase to start. The desired semantics are as follows.

1. A component can't enter phase (**k**+1) until every other component has entered phase **k**.
2. If all components are in phase **k**, then eventually all components should move on to phase (**k**+1).

Here, property **Last** represents the current phase of the components belonging to the group, and is aggregated by computing the minimum over the set of components. Property **Next** represents the new phase to start. Note how our rules ensure that it's never more than 1 phase apart from the phase of the slowest component. When **Next** changes, the Phase() method of a component is called with the new value.

4.3 Atomic Commit

Figure 7 illustrates a leaderless atomic commit protocol, in which the "global" entity plays the role traditionally assumed by the leader. The protocol runs on a stream of messages, on which participates are asked to vote commit or abort. A given message is committed only if all components vote to do so; mechanisms of this sort are common in reliable multicast or database transaction processing. Notice that the code is written in to perform decisions in batches, and the sense in which having the Oracle "around" gives "all" a firm definition, which simplifies PL code. A protocol designer using the PL will often want to write code with the architecture of the PF in mind; the batch commit/abort shown here handles event-streams far better than a "one-by-one" protocol could, and corresponds nicely to the token-based implementation in the PF.

```
protocol TwoPhaseCommit {
  interface { callback Receive(int m);
              bool Ok(int m);
              Commit(int m);
              Abort(int m); }
  properties { intset CommitOk, AbortOk,
                      ToCommit, ToAbort; }
  bindings {
    on Received(m) : if (Ok(m)) CommitOk += m;
                                else AbortOk += m;
    on update ToCommit(add x) :
      foreach (m in x) Commit(m);
    on update ToAbort(add x) :
      foreach (m in x) Abort(m); }
  rules {
    CommitOk :=
      [mono,all]children(∩).CommitOk;
    AbortOk [mono]∪= children(∪).AbortOk;
    global.ToCommit [mono]:= CommitOk;
    global.ToAbort [mono] := AbortOk;
    ToCommit [mono]:= parent.ToCommit;
    ToAbort ∪= parent.ToAbort; } }
```

Figure 7. A simple atomic commit protocol.

5 Architecture

5.1 Translating Specifications to Code

The PF is basically a compiler for PL and a runtime environment with a mixture of generic mechanisms and pre-designed structure. Protocol specifications are converted into .NET [4] code that implements the behavior of protocol "agents", tokens rings etc. The code would normally be compiled statically but could even, if necessary, be generated dynamically. The protocol agent code that emerges from this translation is then loaded into the application using the PF.

The PL compiler is still a work in progress. We have a working compiler that can translate PL codes of the sorts used in the figures here, but some of the optimizations needed to achieve the best possible performance aren't implemented yet. As a result, in the experiments reported later we manually inserted a few additional rules, to express the necessary optimizations that will be handled by the final version of the compiler. Over time, we hope to achieve fully automated translations competitive with hand-coded ones.

Compiled PL code has the form of a data-flow graph together with small fragments of .NET CLR byte code implementing the PL rules. By now, we've seen the main elements of the architecture. The Oracle configures component groups into hierarchical rings, then launches the components, which instantiate protocol agents to play the various roles required by the hierarchy (as leaf nodes, inner entities, and so forth). Components then initiate events, which trigger updates to local copies of properties. As tokens circulate within the various levels of the tree, they sweep up information needed to push the overall group protocol forward, enabling rule transitions in what can be recognized as a traditional data-flow style.

5.2 Extending the Type System

Recall that we cited integration with the .NET type system as a strength of our approach. In what sense is the PF integrated with the .NET CLR, and why is this beneficial?

The .NET component integration system, similar to the component integration system in the Java J2EE environment, is "managed" in several respects. First, a single memory management mechanism is used throughout, making it possible to pass objects from one component to another without copying. Next, languages are compiled into type-safe code. This eliminates many kinds of cross-component overheads, because component boundaries don't need to implementation protection boundaries. There is a single pervasive threads implementation, again capable of supporting cross-component actions. And fi-

[4] Our architecture is OS and language independent, but the prototype was implemented in C# on .NET.

nally, the system provides type checking, again over component boundaries. All of these features combine to improve programmer productivity, ease of debugging, and so forth.

When PF component groups are used within .NET, we essentially introduce a new kind of .NET object – a form of "distributed" object in which the local object types are extended by a new form of distributed type, defined by the PL script associated with the object. For example, if the PL script defines a transactional 1-copy serializability property for some group, we can understand transaction 1-copy semantics as a new kind of distributed type, and similarly for a state-machine replicated object, a virtually synchronous process group, and so forth. In effect, the component type has become a tuple: the type signature for the component itself, and the type signature for the communications group it uses.

We believe that this new kind of distributed type system brings important benefits. Not only can the developer begin to reason about distributed objects much as one reasons about a non-distributed object today, but the runtime system can also enforce type checking automatically, thereby ensuring that a component joining a group is compatible with the group. The system can implement automated type coercions: an object with a weak form of reliability could be "converted" into some stronger form, by replacing weak PL code with PL code that implements a covering behavior (virtual synchrony, for example, is a stronger behavior than best-effort reliability, hence an application correct with best-effort reliability can also work correctly with a virtual synchrony protocol).

5.3 Scalability Considerations

Our other big goal was scalability. Let's revisit the three dimensions enumerated in the introduction and ask how the PF architecture responds to each.

1. **Streams of events.** Recall that in a high performance application, components might generate very large numbers of events each second. The PF addresses this form of scalability in two major ways. First, high-volume data passes through the dissemination side of the PF, which aggregates small messages into large ones and transmits both kinds using fast mechanisms such as IP multicast. Second, control actions are aggregated across potentially large sets of separate protocol instances. We saw an example of this in the commit protocol, which commits or aborts messages in batches. Stream processing is also facilitated by the asynchronous data-flow architecture, which leaves the system flexibility to schedule actions at convenient times (namely, when it wants to circulate tokens).
2. **Number of members.** In large groups, efficient multicast protocols often need hierarchical structures to achieve good performance. We've seen that PL code can express hierarchy in a clean, data-flow manner, although the coding style does take a little getting-used to.
3. **Churn.** The PL treats both failures and joins as special cases of the general case of data-flow execution of the relevant script. For a failure, once the Oracle reports the event to the components in a group (which occurs in a rapid and consistent manner), the failed node is ignored; for a join, a novel asynchronous catch-up mechanism allows a joining component to converge towards consistency with respect to active members, minimizing synchronization delays.

Thus, the PF is successful with respect to these three goals; our experiments confirm the assertions just made. But there is a one more form of scalability on which we should touch. Suppose that the PF is used widely in a distributed system. Distinct applications might now use the PF on the same nodes, creating many component groups running separate protocol instances. We support the obvious option, treating each group separately. But can we do better?

The core technical problem comes down to the encoding of properties into a form that tokens can carry and aggregate efficiently. With multiple groups that overlap *perfectly*, as might occur if some application simply uses multiple groups, with all of its components running the same protocols, the PF implements a simple optimization: it uses a single token hierarchy on behalf of the whole set of protocols, and the token itself becomes a vector, with one field per group. Of course this only works if the vector of per-group information is small enough to fit in the token.

One last case is worthy of mention. Our current system has hand-compiled support for a simple reliable delivery, much like the protocol of Section 3: as long as the Oracle reports that a given component is healthy, other members endeavor to ensure that it receives all messages sent, with no ordering or fancy synchronization. In turns out that this simple form of reliability can scale well: we found a way to perform recovery for many such groups at a time, using a single shared protocol. The token remains small, and its size is unrelated to the number of groups.

Our first release of the PF will provide support for large numbers of protocol instances having the basic reliability property, and will also permit superimposition of a smaller number of protocols with custom PL scripts running on the same nodes. Our belief is that for most applications this will suffice: the basic reliability groups can transport data, while

applications can use a smaller number of protocols with stronger properties for coordination.

6 Experimental Evaluation

In this section, we present simulation results. We run the actual translated protocols within the real PF infrastructure implemented in .NET, but over a virtualized network, and using a simplified prototype version of the Oracle. The only significant difference between the results we present and results that would be obtained from a real deployment in a datacenter boils down to whether the Oracle can be made scalable enough to update system configuration in a timely fashion in the presence of churn and failures. We've designed a hierarchical Oracle that should achieve this, but the scalable version wasn't ready in time for submission[5]. Nevertheless, all aspects of the system that involve running the actual protocol, including the handling of reconfiguration between components orchestrated by the Oracle, are real, and all the observations we make should be valid in general. The discussion is structured around the dimensions of performance and scalability of importance in practical scenarios, and on the overheads PF incurs.

6.1 Processing Events at High Rates

We want to use PF in data centers, financial institutions and other settings where high volumes of events may be generated, and yet a form of global coordination may be needed. For example, servers in a data center might pre-process database transactions, but need to globally coordinate on whether they should be committed. Similarly, if the PF is to replicate components that update state with virtual synchrony multicasts, one would expect that group to handle high volumes of multicasts. How fast can PF run if used within a system with high event rates, and what are the main limitations?

To find out, we compiled the commit protocol of section 4.3 and ran it on a network of **N** = 10,000 nodes. We vary the number of transactions a second and measure the average and maximum time it takes for a node to receive the decision to commit or abort (Figure 8). We extended the code from Figure 7 with

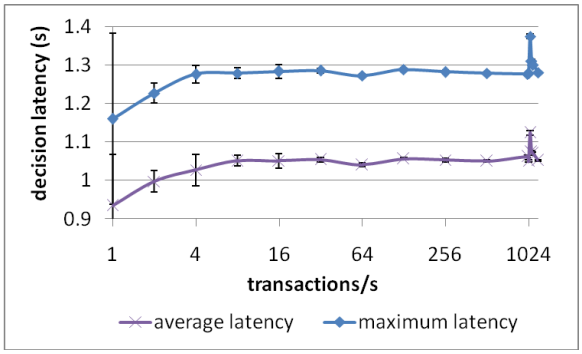

Figure 8. The latency to reach a decision in the commit protocol is insensitive to the transaction rate, but the sustainable transaction rate is limited to about 1024/s due to fixed token rates and sizes. N = 10K, fanout 10, P(commit) 95%, latency 10ms

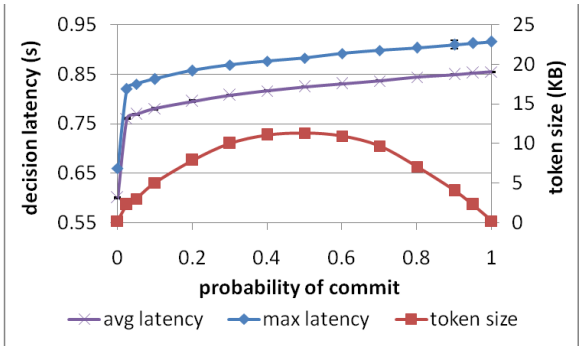

Figure 9. The more unpredictable the decisions of the protocol, the larger tokens needed to fully represent them. Decisions that require agreement (commit) take more time and increase the latency. N = 4096, fanout 8, 1000 TPS, relaxed token sizes

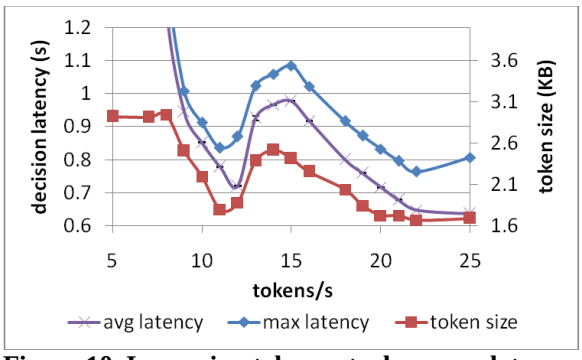

Figure 10. Increasing token rate decreases latency only up to the point where new tokens are released before preceding tokens completed a full round, at which point much redundant work is performed. N = 4096, fanout 8, 1000 TPS, bounded token sizes

a few additional rules to prevent the protocol from voting on transactions that have already been decided (the current compiler doesn't yet automate such optimizations), but we run real PL code, in the real PF.

In this scenario, transactions are delivered to all nodes simultaneously. The Oracle organizes the sys-

[5] The final version of this paper will include experiments of a complete system on real hardware, for real applications in a small datacenter of about 200 physical nodes. At a minimum, we'll have data from our hand-compiled reliable multicast protocol, which achieves extremely high event rates and scales well, side-by-side with the same code as compiled by PF from a script. But we hope to have virtual synchrony, state machine replication and transactions running, and should be able to report performance of each.

tem as a hierarchy with fanout = 10. In each token ring, the PF circulates 10 tokens/s, the sizes of tokens limited to approximately 3KB, to bound the per-node resource usage. Nodes randomly decide to commit or abort; the probabilities are set so that 95% transactions globally commit. Node-to-node unicast latency is uniformly distributed between 9ms and 11ms.

We find that latency is fairly insensitive to the transaction rate. PF can compactly encode information about multiple transactions in a single property. For example, we are able to use simple ranges of integers to represent sets of commit decisions. However, as transaction rate increases, the size of tokens grows, linearly (not shown here). At around 1024 transactions/second, token sizes reach the limit imposed by the PF. At this point, transactions start to pile up, and the latency grows dramatically. This illustrates one limitation: with a fixed token rate and their bounded size, there is a limit to how many such "simultaneous decisions" PF can handle.

The actual limit depends on the nature of information being exchanged. For example, in the commit protocol, it depends on the probability of commit. Transactions are likely to be popular in real uses of the PF, hence we looked closely at token size as a function of this commit/abort ratio (Figure 9). If most transactions commit (or if most abort), PF can represent multiple decisions compactly. If the odds are 50-50, no compression is possible. Transactions seem to be an especially difficulty case: for the other protocols we studied (virtual synchrony, etc), there are simple, compact ways to represent properties.

In general, the latency to make a decision is approximately $2\mathbf{h} / \mathbf{r}$, where **h** is the height of the hierarchy, $\mathbf{h} \geq \log_{\mathbf{f}}\mathbf{N}$, and **r** is the token rate. The capacity of the PF "decision channel" can thus be increased by sending tokens more often (Figure 10), thus reducing the time information about any given transaction circulates within PF. However, the interval between tokens must remain larger than the time for a token to circulate: $\mathbf{r} \leq 1/\mathbf{Lf}$, where **f** is the hierarchy fanout, and **L** is the latency of a node-to-node unicast.

In a data center scenario with network latency on the order of 1ms and the Oracle creating deep hierarchies with fanout **f** = 3 we can circulate 320-330 tokens/s. In a system of **N** = 10,000 nodes, the PF could make decisions in ~50ms (our simulations confirm this), approximately 20 times faster than in scenario on Figure 8, thus permitting much higher event rates.

6.2 Scaling to Very Large Deployments

While data centers are an important scenario for us, we'd like to use PF in much larger systems, such that could span a significant portion of the Internet, for example as a part of a massively multiplayer gaming platform that could involve hundreds of thousands to millions of simultaneous clients scattered over a wide area network and maintain a tremendous amount of state that couldn't be handled by a single server farm, but that may need to be kept consistent.

Could the PF efficiently support large deployments? How does performance degrade with scale?

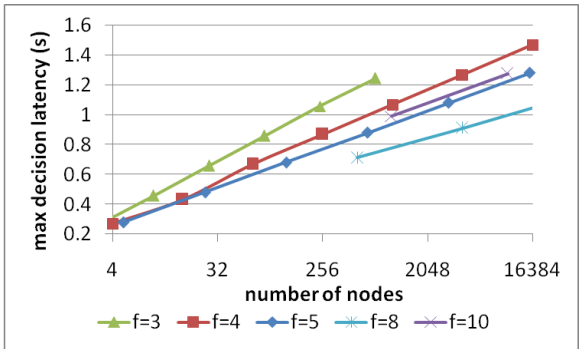

Figure 11. Decision latency in the commit protocol as a function of system size, with varied fanout (f). 1000 TPS, P(commit) 95%, network latency 10ms

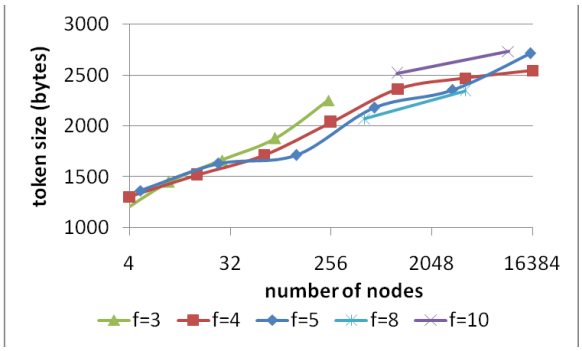

Figure 12. Token size grows logarithmically with system size. This, in addition to the growth in decision latency, limits achievable scalability for any fixed maximum token size.

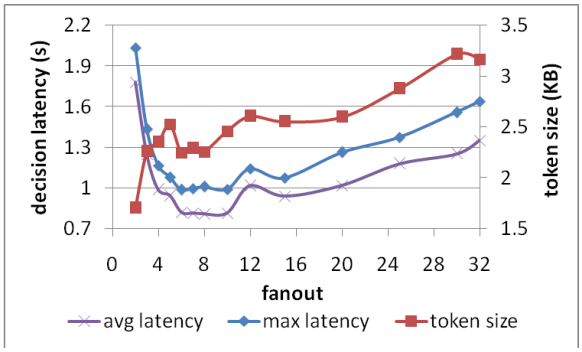

Figure 13. With the token rates constant, larger fanouts result in higher token roundtrip times, but small fanouts create deep hierarchies, and both can hurt latency. Thus, the performance of PF depends on being able to fine-tune token rates to ring sizes and network latency.
N = 1000, 1000 TPS, net latency 10ms, 10 tokens/s

In the preceding section, we saw that PF's "information channel" has a limited capacity. Once it is saturated, the number of events per second PF can process is inversely proportional to the decision latency. Hence, latency is a key performance metric.

With a fixed fanout, the depth of the hierarchy, and hence the decision latency, grows logarithmically with system size (Figure 11), and hence is asymptotically optimal. However, so does the token size because transactions take longer to decide (Figure 12), thus placing the upper limit on the size of the system that can handle the given workload. With our 10ms network latency, 1000TPS, 95% commit probability and the tokens limited to about 3KB, PF can scale to about 20,000 nodes. Scaling further would require faster hardware, larger tokens, or a lighter workload.

In a large system, particularly in a wide-area network, it may also be harder to optimally configure the system. The PF is sensitive to token rates (Figure 10), and to fanout (Figure 13). It's easy to verify that best performance could be achieved if Oracle were able to maintain a deep hierarchy, with a fanout 3..5, and if tokens circulated without stopping.

6.3 Handling Node Crashes and Churn

In large systems, particularly with deeper hierarchies, churn is an issue. In data center scenarios, it is usually negligible except after major power outages. However, as mentioned, we would like to support mobile agents using interactive applications, massively multiplayer gaming platforms, and other applications where users might stay subscribed for periods of time on the orders of minutes or less.

How does frequent reconfiguration affect PF performance?

In PF, a crash may result in a loss of high-level state, disrupting the system until the Oracle reconfigures that level and state can be reconstructed. However, PL is designed to make protocols expressed in it immune to a temporary loss of state, and experiments confirm that in practice, PF handles churn very well.

To isolate the pure effect of churn on the speed at which the PF makes decisions, we run the CoordinatedPhases protocol of section 4.1. The nodes independently crash and reboot according to exponential distributions. To maximally stress the system, we set the average time to reboot (MTTR) to 5s and we vary the average time to failure (MTTF) from 1s to 100s.

As we have already argued, the time to make decisions (in this case the interval between subsequent phases) is the key factor that affects the performance of PF. Figure 14 looks at extreme churn rates, where each node crashes after 10s, and reboots in 5s, and 66% of the system is down at any given moment. Nonetheless, even for a very large 32768-node system, performance drops by only 20% as compared to

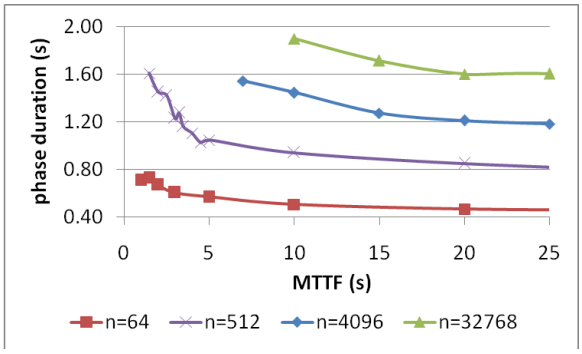

Figure 14. The duration of one phase in the protocol of section 4.1, for varying system sizes (n) and mean time to node failure (MTTF). Even extreme churn has little impact on the performance on PF. With MTTF less than a few seconds, the system is reconfiguring itself too much to do a useful work. MTTR 5s, fanout 8, net latency 10ms, 10 tokens/s

the case where no failures occur. After reconfiguration, it typically takes 2-3 token rounds to bring new or reincarnated agents up to speed, renegotiate token contents, and reconcile the versions of property values. All progress then resumes at full speed. With MTTF on the order of seconds, each ring still spends more time on useful work than reconfiguring itself.

The protocol we evaluated is simple, but more complex protocols scale with churn in just the same

Table 1. Time decomposition of token processing times. Serialization and deserialization of packets with tokens accounts for 75-99% of the overhead.

number of protocols	1	10	100
Coordinated Phases (section 4.1)			
aggregation (µs)	1	9	101
dissemination (µs)	1	8	106
serialization (µs)	92	561	5105
Commit (section 4.3)			
aggregation (µs)	16	240	couldn't simulate
dissemination (µs)	38	440	
serialization (µs)	147	1247	

Table 2. Token sizes and processing times include everything from a deserialization of the incoming packets, to the serialization of outgoing packets. N = 3125, fanout 5, net latency 10ms, 18 tokens/s, P(commit) 95%, 1000TPS. With 100 commit protocols the simulator crashes (insufficient memory).

number of protocols	1	10	100
Coordinated Phases (section 4.1)			
token size (KB)	0.10	0.69	6.53
time per token (µs)	94	578	5312
Commit (section 4.3)			
token size (KB)	1.31	13.71	couldn't simulate
time per token (µs)	197	1927	

way; they simply carry more state in tokens. In any real system, performance with churn will be dominated by the speed at which joining protocol participants can load the state from existing members, and the stress this places on the network. This depends on the application, and is completely independent of PF.

We note that performance of the PF could be improved with a mechanism to smoothly "migrate" state from nodes that are leaving but that haven't crashed.

6.4 Network and Processing Overheads

The processing overheads in PF are fairly small. In the absence of failures, the only overhead is that of processing the tokens. For a single protocol instance, the time to receive, deserialize, process, serialize and send tokens is on the order of 100-200μs (Table 2). When multiple protocol instances are active, the aggregative structure of PL scripts provides efficiencies: tokens carry information on behalf of multiple instances, and rules can be executed in batches.

The decomposition of processing times (Table 1) shows that the actual protocol logic (further decomposed to logic for property aggregation and dissemination) is cheap to execute. Serialization and deserialization are responsible for 65-99% of the overhead. The serialization-related costs could be reduced if we implemented token processing in unmanaged C++. Our current implementation translates specifications in PL to C# code that runs in the same .NET runtime. The amount of generated code is fairly large (1100 lines for CoordinatedPhases, 1300 lines for Commit), but the generated code is fairly simple, and consists of a number of assignments, application of binary operators, and comparisons on versions of property values, none of which uses sophisticated language or runtime support and could be easily expressed at a low level. Indeed, if we run into situations where performance is a problem, PL code could be compiled directly to machine instructions.

7 Conclusions

We've demonstrated a novel approach to building distributed, reliable protocols, by modeling their logic as distributed, asynchronous information flows, within a hierarchical structure managed by an external, fault-tolerant oracle service. The PL is expressive enough for most practical purposes. Although brevity precluded a detailed discussion of all language mechanisms and a complete presentation of protocols such as virtual synchrony or 3PC, all of these are efficiently expressible in PL. Although the PL takes some getting used to, protocols are easier to understand, debug and reason about than many prior approaches. By integrating the PF with .NET, we are able to support a new kind of distributed live object, in which object replication semantics are determined by PL code. The connection between PL scripts and type systems is especially promising.

The PL compiler and PF infrastructure are (mostly) complete, and experiments confirm that the approach can achieve high performance, scalability and immunity to churn.

8 References

[1] M. Abadi, B. Loo. Towards a Declarative Language and System for Secure Networking. 3rd International Workshop on Networking meets Databases (NetDB), Cambridge, MA, Apr 2007.

[2] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Journal of the ACM*, 43, 4 (Jul. 1996), 685-722.

[3] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. ACM Transactions on Computer Systems, Vol. 20, No. 3, August 2002, p. 191-238.

[4] L. Lamport. The Temporal Logic of Actions. In ACM Transactions on Programming Languages and Systems, 16, 3 (May 1994), 872-923.

[5] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In the Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005).

[6] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. CWI-Quarterly, 2(3):219--246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[7] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A Framework for Protocol Composition in Horus. In proceedings of Principles of Distributed Computing (PODC '95).

[8] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004).