Achieving Reliability Through Distributed Data Flows and Recursive Delegation

Krzysztof Ostrowski[†], Ken Birman[†], Danny Dolev[§], and Chuck Sakoda[†]

[†]Cornell University
Ithaca, NY 14850, USA
{krzys, ken, cms235}@cs.cornell.edu

[§]Hebrew University
Jerusalem 91904, Israel
dolev@cs.huji.ac.il

Abstract

Strong reliability properties, such as state machine replication or virtual synchrony, are hard to implement in a scalable manner. They are typically expressed in terms of global membership views. As we argue, global membership is non-scalable. We propose a way of modeling protocols that does not rely on global membership. Our approach is based on the concept of a distributed data flow, a set of messages distributed in space and time. We model protocols as networks of such flows, constructed through recursive delegation. The resulting system uses multiple small membership services instead of a single global one while still supporting stronger properties. Our work was inspired by the functional approach to modeling distributed systems pioneered by I/O automata. This paper focuses on the basic model. Internal details of our system architecture and a compiler that translates protocols from our data flow language to real executable code will be discussed elsewhere.

Keywords: strong reliability properties, I/O automata, distributed data flow, scalable protocol, monotonic aggregation

1. Introduction

We believe there is a need for large-scale replication and reliable multicast; for example, in data centers, reliable multicast groups including thousands of nodes could store dynamic configuration state, such as partitioning of resources across applications. They could be used to *consistently* and *reliably* distribute security policy updates, keys, or patches, thus enabling fast and well-coordinated response to a threat.

Our work is focused on distributed replication protocols, in which nodes can dynamically join, leave, or fail by crashing, and where churn could be high. This is typical of peer-to-peer scenarios, but it can also happen in data centers during load surges: timing-out connections can be perceived as failures. Churn tolerance is key to stability in such systems.

The term "strong properties" in this paper refers to quasi-absolute [4, 7] properties such as virtual synchrony, atomic broadcast, commit, transactions, state machine replication, or consensus with dynamic membership. In the current implementations of protocols that offer such properties, nodes are controlled by membership views generated by a *global membership service* (GMS), which can be external, or a part of the protocol itself, and strong guarantees are expressed in terms of the global views. Our work wouldn't be applicable to, for example, gossip protocols: these avoid global views, but whereas we achieve strong properties, they typically are limited to weaker (convergent) ones.

GMS-mediated reliability can be problematic. First, as a system grows in size, the frequency of membership changes grows proportionally; eventually, this can become a serious burden on the members. Second, in protocols such as virtual synchrony every change triggers $\mathcal{O}(n)$ work: an $\mathcal{O}(n^2)$ cost that rapidly becomes prohibitive. Batching changes is an obvious option to consider, but doing so raises other concerns: in many protocols, progress can't occur if a member becomes unresponsive until the GMS excludes it from the view. These factors compel a rethinking of the relationship between membership and reliability, and suggest that globally visible, consistent, and incrementally reported membership should not be a part of large-scale reliability models.

We propose a new approach, in which strong properties emerge in the network in a decentralized, hierarchical manner, and the role traditionally played by a single, monolithic GMS is hierarchically decomposed and blends into the hierarchy established by a scalable protocol.

Our technique relies on a new concept of monotonic aggregation, which records all progress made by the protocol in a hierarchical distributed flow of values that describe system state, global and local decisions. Membership changes are viewed as perturbations that disrupt the integrity of this flow. The flow is designed to tolerate such perturbations by locally recreating information that has been recorded in it.

This paper makes the following contributions.

- We propose a new model that allows the global behavior of a distributed protocol to be expressed in a purely functional style, as a graph of distributed functions that operate on sets of messages spread across the network.

We introduce basic building blocks, and illustrate their use by dissecting a simple reliable multicast protocol.

- We explain how strong properties map to the properties of our basic building blocks; in particular, we explain how *monotonicity*, a core concept in our approach, can be used to reliably record protocol state. We introduce and prove theorems characterizing the conditions under which monotonicity can be achieved using simpler properties, and how it can be hierarchically composed.
- We present an architecture that allows flow hierarchies to be created through recursive delegation and employs multiple small membership services to support a large group of clients. Our approach has the additional benefit of decoupling the core protocol semantics from the process of establishing and maintaining the hierarchy.
- We discuss initial performance results with a real protocol stack running on top of a discrete event simulator, focusing on two key factors affecting the performance of monotonic aggregation. The results suggest that our approach can scale well and tolerate high rate of churn.

Our prototype system, called Quicksilver Properties Framework, includes a compiler that translates protocol specifications from data flow notation similar to that on Figure 3 into executable code, but due to limited space we omit details of the language and system internals, and we focus on the underlying model. Our goal in this paper is to show, through analysis and simulations, that distributed monotonic aggregation is a viable alternative to GMS in terms of being able to support strong properties.

Due to the space constraints, we present only those definitions, theorems, proofs, and elements of the supporting architecture, that are essential to understanding our approach.

2. Model

2.1. Distributed Flows

We define a (*distributed data*) *flow* as a set of quadruples of the form (x, t, k, v); each quadruple represents a *message* carrying value $v \in \mathcal{V}$ tagged with a *version* number $k \in \mathcal{K}$, exchanged on node $x \in \mathcal{N}$, at time $t \in \mathcal{T}$, by instances of two software components or protocol layers. Messages at the same node x always flow at different times, and if they carry the same version k, they also carry the same value v. $\mathcal{T}$ and $\mathcal{K}$ must be isomorphic[1] with $\mathbb{N}$ (natural numbers).

We focus mainly on *control* flows, where each value represents a protocol state or a decision. To support protocols that stream data at high rates, we focus on batched processing, where each value can carry state or decision regarding many application events at once, and processing at all layers in the protocol is done for sets of such events in parallel.

For example, in reliable multicast we can identify three flows R, F, and D, defined as follows. Each $(x, t, k, v) \in R$ represents a method call from a layer that receives packets from the network to upper layers that implement loss recovery. It is the k-th such call on x, and we can logically think of v as representing the set of identifiers of all packets that x received until time t, e.g., $v = \{1..25, 28\}$ (note how we use a set notation to enable batched processing). For notational convenience, we denote this value as $R_x(k)$ (see Figure 1).

We assumed that $R_x(k)$ contain identifiers of *all* packets, but only for *modeling* purposes. Our methodology permits the introduction of optimizations (such as truncating $R_x(k)$ to contain only identifiers not previously reported) at a com-

[1]The model can be extended to use partial ordering on $\mathcal{T}$. We use total ordering for clarity of the presentation. All proofs carry over without much change. The timestamps are hidden and not accessible to any of the nodes.

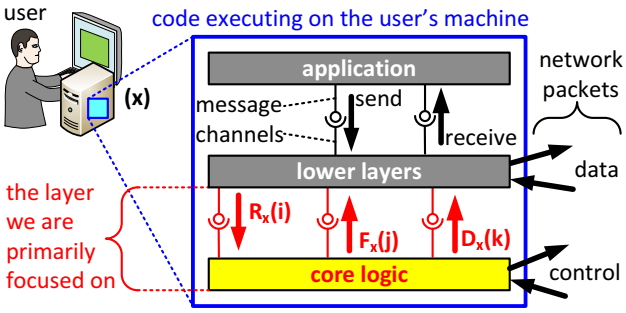

Figure 1. Multicast logic as a transformation from inputs $R_x(i)$ to outputs $F_y(j)$ and $D_z(k)$.

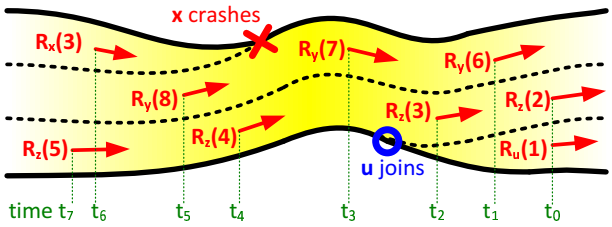

Figure 2. Messages in flow R are distributed in time and space (different k and x in $R_x(k)$).

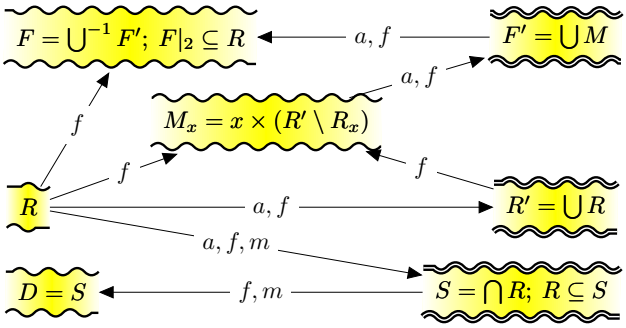

Figure 3. Multicast logic modeled as a graph of transformations on distributed data flows.

pilation stage, while transforming specifications into code.

Similarly, each message in D carries a value $D_x(k)$, the set of identifiers of packets that can be delivered to the application; values $D_x(k)$ flow from the layers implementing the core synchronization logic to a lower layer that manages the receive buffers. Finally, messages in F model requests to forward packets to other nodes. Each $F_x(k)$ is a set of pairs (y, i), where y is the destination that x should forward data to, and i is the identifier of the packet to be forwarded.

Messages in a given flow are *distributed* in space (x) and time (t). The set of nodes on which messages are appearing often changes as nodes join or leave the protocol (Figure 2).

In every protocol, we can distinguish the part of the stack that implements the core logic, whereas other parts might be involved in tasks such as physical network transmissions or buffering. In our approach, we express the behavior of this core logic as a distributed *function* generating output flows from input flows; specifically, each value in an output flow is expressed as a result of applying some operator to some set of values at the input. In our example, R is an input flow; values in D, F are calculated from values in R. Each value is computed from past input; i.e., a value flowing at time t can only be computed from values flowing at times $t' < t$.

Our protocol as a function is *distributed* in the sense that a given value $D_x(i)$ or $F_x(i)$, generated on node x, depends not only on values $R_x(j)$ received on x, but usually also on many values $R_y(k)$ received on multiple other nodes $y \neq x$.

Like I/O automata (IOA) [17], our model is functional, and it may be possible to express it as an extension of IOA. However, it also differs from IOA, in that its purpose is not only to specify protocol behavior in terms of events, but to do so constructively, and in a manner that represents the logical flow and can be automatically translated into a scalable implementation. To this end, we introduce a set of scalable building blocks – *aggregations*, *transformations*, *disseminations*, and *distributions* – and we show how to express the semantics of multicast as compositions of these (Figure 3).

2.2. Building Blocks

Aggregations. Flow β is an *aggregation* on flow α if for some associative commutative binary *operator* $\otimes : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$, and a set of finite *membership* functions $\mu_x : \mathcal{K} \to \mathbb{P}(\mathcal{N})$ and *selector* functions $\sigma_x : \mathcal{K} \times \mathcal{N} \to \mathcal{K}$ for all $x \in \mathcal{N}$, the following condition holds for each message $(x, t, k, v) \in \beta$:

$$\beta_x(k) = \bigotimes_{y \in \mu_x(k)} \alpha_y(\sigma_x(k, y)) \ . \tag{1}$$

Intuitively, condition (1) states that each value appearing in β can be represented as a result of applying operator $\otimes$ to a set of values appearing in α. Function μ_x determines nodes from which values are taken, and functions σ_x determine the versions of values appearing at those nodes that must be used in the aggregation. Values in β with different versions k, at different locations x, may be aggregated over different sets of nodes $\mu_x(k)$, and different versions $\sigma_x(k, y)$. Placing additional constraints upon μ_x and σ_x allows us to distinguish between different "flavors" of aggregation, such as *in-order* or *guarded* aggregations we discuss later.

In our reliable multicast example, we can identify an internal flow S defined as an aggregation over R using the set intersection operator $\cap$ (see Figure 3, bottom-right). Each value $S_x(k)$ is a set of identifiers of packets *stable* in some group $\mu_x^S(k)$, i.e., received by every node in it (note that we used the superscript S to refer to the particular membership function defining S). In many protocols, senders compute $S_x(k)$ by collecting ACKs to drive retransmissions, and hierarchical protocols, such as RMTP, compute $S_x(k)$ for increasing subsets of nodes $\mu_x^S(k)$. We do not assume any particular implementation or method of aggregation; we simply note that S materializes at a certain layer in the system.

Similarly, we can identify flow R', an aggregation on R using the set union operator $\cup$. Each $R'_k(x)$ carries information about messages received by some nodes, in a certain group. Multicast protocols often collect such information to issue NAKs to the sender, or to drive peer-to-peer recovery.

It should be noted at this point that although equation (1) does refer to memberships $\mu_x(k)$, we are only assuming the *existence* of such sets. Unlike in the traditional approaches, where nodes must actually *learn* global membership as part of the protocol, in our architecture memberships $\mu_x(k)$ are never explicitly constructed and never materialize anywhere in the system. We discuss this in more detail in Section 2.5.

The definition of flow S on Figure 3 contains an equation $R \subseteq S$. We use such equations to further constrain the way values must be aggregated; we discuss them in Section 2.4.

Transformations. Flow β is a *transformation* of flows $\alpha^1, \alpha^2, \ldots, \alpha^n$ if for certain operators $\Psi_x : \mathcal{V}^n \to \mathcal{V}$, membership functions $\mu_x : \mathcal{K} \times \{1, 2, \ldots, n\} \to \mathcal{N}$, and selector functions $\sigma_x : \mathcal{K} \times \{1, 2, \ldots, n\} \to \mathcal{K}$, where $x \in \mathcal{N}$, the following holds for each message $(x, t, k, v) \in \beta$:

$$\beta_x(k) = \Psi_x(v_1, v_2, \ldots, v_n) \ , \tag{2}$$

$$\text{where} \quad \forall_{1 \leq i \leq n} \ v_i = \alpha^i_{\mu_x(k,i)}(\sigma_x(k, i)) \ . \tag{3}$$

Intuitively, these conditions state that every value in β is a result of applying Ψ to a list of values from $\alpha^1, \alpha^2, \ldots, \alpha^n$. Again, we can distinguish between different flavors of transformations by placing additional constraints on μ_x and σ_x.

In multicast, we can identify an internal flow M, a transformation on R_x, R' defined by $\Psi_x^M(v_1, v_2) = x \times (v_2 \setminus v_1)$. Here, component $v_2 \setminus v_1$ represents the set of messages missing at node x, calculated based on the most recent values v_1, v_2 obtained from flows R_x, R', and $x \times (v_2 \setminus v_1)$ represents the set of forwarding requests that must be satisfied for node x to catch up with other nodes. These values are aggregated to form an internal flow F' representing a global "todo" list.

In a real system, M could be represented by NAKs and aggregated into F' at the sender, and in peer-to-peer protocols, different nodes x might calculate their $F'_x(k)$ independently.

Even if forwarding requests are not centrally aggregated, it is still useful to think of S, R', and F' as representing the global state of the system (this is symbolized by the double border on Figure 3). In Section 2.5, we show how to implement these *global* flows in a scalable, hierarchical manner.

Disseminations. Flow β is a *dissemination* of α if each value appearing in β appeared previously in α; formally, the following holds for some $\mu_x : \mathcal{K} \to \mathcal{N}$ and $\sigma_x : \mathcal{K} \to \mathcal{K}$:

$$\beta_x(k) = \alpha_{\mu_x(k)}(\sigma_x(k)) \ . \tag{4}$$

Dissemination also has multiple flavors depending on which values from α are included in β, where, or in what sequence.

In our example, flow D is a dissemination of S; meaning that our protocol is delivering packets stable on sets $\mu^S_x(k)$.

Distributions. The concept of *distribution* can be understood as the opposite of aggregation; flow β is a distribution on α if each value v in a subset $\alpha' \subseteq \alpha$ maps to a set $V \subseteq \mathcal{V}$ of values in β such that aggregating values from V yields v. Formally, we assume the existence of *distribution* functions $\delta_x : \alpha' \to \mathbb{P}(\beta)$, for $x \in \mathcal{N}$, for which the following holds:

$$\forall_{m \in \alpha'} \ \nu(m) = \bigotimes\nolimits_{m' \in \delta(m)} \ \nu(m') \ , \tag{5}$$

$$\beta = \bigcup\nolimits_{m \in \alpha'} \ \delta(m) \ . \tag{6}$$

In the above, $\nu(m)$ is the value of m, $\nu((x,t,k,v)) \triangleq v$.

Given δ, we can define the aggregation that this distribution is an inverse of by expressing μ and σ in terms of δ. We can then specify the flavor of our distribution by specifying the flavor of this aggregation (details omitted for brevity).

In our example, F is a distribution over F'; global "todo" lists of forwarding requests are partitioned among recipients so that those requests can be handled in parallel. Conditions (5, 6) ensure that local forwarding decisions (F) at all times reflect the combined needs of the system as a whole (F').

Like aggregations, distributions can also have additional conditions, limiting how much of the value being distributed can be "chopped off" and placed in any individual message. In our example "$F|_2 \subseteq R$", which is as abbreviation for the longer $\forall_{x \in \mathcal{N}} \forall_{k \in \mathcal{K}} \exists_{k' \in \mathcal{K}} \ \{i \mid \exists_y \ (y,i) \in F_x(k)\} \subseteq R_x(k')$, states simply that a node cannot commit to forwarding packets that it has not already received.

2.3. Strong Semantics

In our model, strong semantics are expressed by defining a predicate $\phi : \mathcal{V} \to \mathbb{B}$, where $\mathbb{B}$=$\{true, false\}$, and stating that if $\phi(v)$ holds for some v at node x in some flow α, then for each non-faulty node $x_0 \in \mathcal{N}_0$, some value v_0 eventually flows at x_0 in α such that $\phi(v_0)$ holds, as specified below:

$$\forall_{m \in \alpha} \ (\phi(\nu(m)) \Rightarrow \forall_{x_0 \in \mathcal{N}_0} \exists_{k_0 \in \mathcal{K}} \ \phi(\alpha_{x_0}(k_0))) \ . \tag{7}$$

A node is *non-faulty* if it eventually begins and never ceases to execute the protocol. We assume the *fail-stop* model [20].

In our multicast example, we would like to guarantee that if any node, even a faulty one, delivers packet i, eventually all non-faulty nodes do so. We can express this requirement by substituting $\alpha = D$ and $\phi(v) \equiv (i \in v)$ in equation (7).

In an asynchronous system, of course, such property cannot be guaranteed unconditionally [7]; typically, it is conditional on the existence of an appropriate failure detector [4]. In practice, the latter is typically approximated by the GMS. The property is then achieved by recording the information about packet i on all nodes in some global membership view before any node can deliver it, and transferring state to new members. This ensures that at the time i is being delivered, information about it has been *remembered*, in the sense that it will reliably affect future decisions made by the protocol.

The key to understanding our approach lies in the different way information is *remembered* in the system. Instead of relying on global views and state transfer, we require that certain flows be *monotonic*, as defined below. Monotonicity itself does not imply equation (7), but it does so in a combination with liveness properties.

Monotonicity. Flow β is (*strongly*) *monotonic* if messages with higher versions also have larger values (with respect to a partial order $\leq$ on $\mathcal{V}$), as formally defined below:

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

In the above, $\kappa(m)$ is the version of m, $\kappa((x,t,k,v)) \triangleq k$.

Note that equation (8) doesn't assume that messages m, m' flow at the same location; if equation (8) holds only for such pairs of messages, β is said to be *weakly* monotonic.

Strong monotonicity is a stronger property, and in particular, it subsumes total ordering: a strongly monotonic flow is *consistent*, where a *consistent* flow is one in which messages carrying identical versions also carry identical values:

$$\forall_{m,m' \in \beta} \ (\kappa(m) = \kappa(m') \Rightarrow \nu(m) = \nu(m')) \ . \tag{9}$$

In our example, R is weakly monotonic and not consistent, for different nodes x generate their $R_x(k)$ values independently. Flow S, however, must be monotonic (this is symbolized by the letter m on the arrow connecting R and S).

Before explaining how to achieve monotonicity, let's illustrate its role in a proof that non-faulty nodes deliver the same packets (equation (7) for $\alpha = D$ and $\phi(v) \equiv (i \in v)$).

Proof (sketch). Suppose that node $x \in \mathcal{N}$ delivers packet i, so $i \in D_x(k)$ for some $k \in \mathcal{K}$. We want to prove that each non-faulty node $x_0 \in \mathcal{N}_0$ eventually also delivers i, i.e., that for each such x_0, there exists $k_0 \in \mathcal{K}$ such that $i \in D_{x_0}(k_0)$.

Since D is a dissemination of S, all values appearing in D must have previously appeared in S, so there exist some $x' \in \mathcal{N}$, $k' \in \mathcal{K}$ for which $D_x(k) = S_{x'}(k')$. Now, if our system doesn't grind to a halt and flow S doesn't terminate,

new messages will keep flowing in S with versions $k'' \geq k'$. At this point, monotonicity of S (with respect to the partial order defined by $\subseteq$) ensures that $S_{x'}(k') \subseteq S_{x''}(k'')$; hence, $i \in S_{x''}(k'')$. If the system is live, some of these values are eventually propagated to flow D at each non-faulty x_0, i.e., $D_{x_0}(k_0) = S_{x''}(k'')$ for some $x'' \in \mathcal{N}$ and some $k_0, k'' \in \mathcal{K}$ such that $k'' \geq k'$. This yields $i \in D_{x_0}(k_0)$. ■

For a full proof, we'd need to formalize progress conditions, ideally using the $\diamond\mathcal{W}$ failure detection model [4]. This is hard [5], and is beyond the scope of this paper.

The reader may have noticed that failure handling is not explicit in our model; indeed, it is implicit in the definitions of properties such as monotonicity. If node x fails right after a value $\beta_x(k)$ flows at it, monotonicity still constraints values $\beta_y(j)$ that flow at nodes $y \neq x$. Protocols implementing monotonicity must explicitly address such cases. Indeed, as explained in Section 3, each flow in our system is internally implemented by a small group of clients managed by a local membership service. Each such group individually handles its own failures; each flow is thus *individually* subject to the FLP result [7], so a flow cannot be both monotonic and live.

2.4. Achieving Monotonicity

The key result of this section is Theorem 2.1, which suggests how to achieve monotonicity using simpler properties, and directly motivates the protocol presented in Section 3.2. Before stating the theorem, we introduce a few definitions.

Aggregation β is *in-order* if functions $\mu_x^\beta, \sigma_x^\beta$ are same for all $x \in \mathcal{N}$ for which they're defined, and if in addition, newer aggregations select newer versions, as defined below:

$$\forall_{x,y \in \mathcal{N}} \forall_{k,k' \in \mathcal{K}} \ (k \leq k' \Rightarrow \sigma_x(k,y) \leq \sigma_x(k',y)) \ . \quad (10)$$

This property is easy to satisfy if aggregation on β happens in rounds, and each node always contributes the latest value.

A commutative binary operator $\otimes$ is *weakly monotonic* if it satisfies only the first, and *strongly monotonic* if it also satisfies the second condition below, for all $v_1, v_2, v_3 \in \mathcal{V}$:

$$v_1 \leq v_2 \Rightarrow v_1 \otimes v_3 \leq v_2 \otimes v_3 \ , \quad (11)$$

$$v_1 \otimes v_2 \leq v_1 \ . \quad (12)$$

Many operators satisfy the above, but with respect to different orders on $\mathcal{V}$, e.g., $\cap$ is such for $v \leq v' \equiv v \subseteq v'$, but for $\cup$, the opposite order must be employed, $v \leq v' \equiv v' \subseteq v$.

Aggregation β on α is *guarded* if for each pair of aggregated values with subsequent versions $k < k'$, and for each new member y participating only in the newer aggregation, the value $\alpha_y(\sigma_{x'}(k',y))$ contributed by y is no smaller than either the full or any partial result of the former aggregation. Formally, for each $k < k'$ such that $\neg\exists_{k''} \ k < k'' < k'$, and each $y \in \mu_{x'}(k') \setminus \mu_x(k)$, the following holds:

$$\exists_{N_p \subseteq \mu_x(k)} \big(\alpha_y(\sigma_{x'}(k',y)) \geq \bigotimes_{z \in N_p} \alpha_z(\sigma_x(k,y))\big) \ . \quad (13)$$

This means that each node joining the aggregation must obtain at least a partial result of the current or the immediately preceding aggregation before its own value can be included.

Theorem 2.1 *If β is a guarded, in-order aggregation on α, using a strongly monotonic, idempotent operator $\otimes$, and α is weakly monotonic, then β is strongly monotonic.*

Proof. Let $\beta_x(k), \beta_{x'}(k')$ be any two values that flow in β such that $k \leq k'$. We need to show that $\beta_x(k) \leq \beta_{x'}(k')$.

Let's partition nodes involved in aggregations into three groups: let $N_o = \mu_x(k) \setminus \mu_{x'}(k')$ be nodes involved in only the older aggregation, $N_n = \mu_{x'}(k') \setminus \mu_x(k)$ those only in the newer one, and $N_b = \mu_x(k) \cap \mu_{x'}(k')$ those in both.

Let $v_o = \bigotimes_{y \in N_o} \alpha_y(\sigma_x(k,y))$ be a partial result of the older aggregation on N_o and $v_b = \bigotimes_{y \in N_b} \alpha_y(\sigma_x(k,y))$ on N_b; the aggregate value is then $\beta_x(k) = v_o \otimes v_b$. Similarly, we have $\beta_{x'}(k') = v'_b \otimes v'_n$ for $v'_b = \bigotimes_{y \in N_b} \alpha_y(\sigma_{x'}(k',y))$ and $v'_n = \bigotimes_{y \in N_n} \alpha_y(\sigma_{x'}(k',y))$. What we need to prove can now be rewritten as $v_o \otimes v_b \leq v'_b \otimes v'_n$. We prove it by showing that (a) $v_b \leq v'_b$, and (b) $v_o \otimes v_b \leq v'_n$. Specifically, we prove it through the following chain of inequalities:

$$v_o \otimes v_b \overset{(i)}{=} v_o \otimes v_b \otimes v_b \overset{(ii)}{\leq} v_o \otimes v_b \otimes v'_b \overset{(iii)}{\leq} v'_n \otimes v'_b \ . \quad (14)$$

In the above, (i) follows from the idempotence of $\otimes$. Then, (ii) and (iii) follow from (a) and (b), respectively, combined with the monotonicity of $\otimes$. We assumed that N_o, N_b, and N_n are non-empty. $N_b \neq \emptyset$ holds because β is guarded. For $N_n = N_o = \emptyset$, the desired result follows from (a) alone. If only $N_n = \emptyset$, then $v_0 \otimes v_b \leq v_b$ follows from strong monotonicity of $\otimes$, and then $v_b \leq v'_b$ from (a). Finally, if only $N_o = \emptyset$, (b) reduces to $v_b \leq v'_n$ as a special case, but the reasoning behind it remains the same as below.

Part (a). Since $k \leq k'$, and β is an in-order aggregation, $\sigma_x(k,y) \leq \sigma_{x'}(k',y)$. Since α is weakly monotonic, we get $\alpha_y(\sigma_x(k,y)) \leq \alpha_y(\sigma_{x'}(k',y))$. Since $\otimes$ is monotonic, we can combine inequalities for all $y \in N_b$; this yields $v_b \leq v'_b$.

Part (b). The fact that β is a guarded aggregation implies that for every $y \in N_n$ there exists $N_p^y \subseteq \mu_x(k) = N_o \cup N_b$ such that $\bigotimes_{y \in N_p^y} \alpha_y(\sigma_x(k,y)) \leq \alpha_y(\sigma_{x'}(k',y))$. Merging these inequalities for all $y \in N_n$ yields v'_n on the right side. Since $\otimes$ is idempotent, the left side becomes an aggregation over $N_p = \bigcup_{y \in N_n} N_p^y$. Now, since $N_p \subseteq \mu_x(k)$ and $\otimes$ is strongly monotonic, so excluding values from $\mu_x(k) \setminus N_p$ could only have made the result larger, the left side is larger than $v_o \otimes v_b$. Thus, we can conclude that $v_o \otimes v_b \leq v'_n$. ■

In our example, R is weakly monotonic, but not consistent, and operator $\cap$ is strongly monotonic and idempotent. The theorem guarantees that S is monotonic *if only* aggregation on R is in-order and guarded. As explained in Section 2.3, this suffices to achieve atomic delivery semantics.

2.5. Hierarchical Composition

To conclude the presentation of our model, we now turn to scalability. For the sake of brevity, we focus on aggregations, as the most interesting (and challenging) case. Other flow types can be handled in a similar manner. The primary result is of this section is Theorem 2.2, which underpins our hierarchical delegation approach presented in Section 3.1.

A set of flows H is an *aggregation network* if there exists a well-founded strict partial order $<$ on H, such that every non-minimal element $\beta \in H$ is an aggregation on the union of its *children*, where the *children* of a flow β, denoted $\mathcal{C}(\beta)$, are its direct predecessors, formally defined as follows:

$$\mathcal{C}(\beta) = \{\alpha \in H \mid \alpha < \beta \wedge \neg\exists_{\gamma \in H}\ \alpha < \gamma < \beta\}\ . \quad (15)$$

We assume that all aggregations are using the same operator and are of the same flavor (e.g., all of them are monotonic). The minimal elements in an aggregation network are called *sources*, the maximal elements are called *sinks*, and the sets of sources and sinks are denoted $\bot^H$ and $\top^H$, respectively.

We assume potentially *infinite* networks, in which a non-minimal β can have infinitely many children, $|\mathcal{C}(\beta)| = \infty$.

Theorem 2.2 *Each sink in an aggregation network H is an aggregation over the union of all sources $\bigcup \bot^H$. If in addition, all sources are weakly monotonic, and all non-minimal $\beta \in H$ are guarded, in-order aggregations on their respective $\bigcup \mathcal{C}(\beta)$ using a strongly monotonic, idempotent operator $\otimes$, then all sinks $\beta \in \top^H$ are strongly monotonic.*

Proof. Let $\beta_x(k)$ be any value appearing in a certain sink $\beta \in H$. We'll construct a value tree, with node $\beta_x(k)$ at the root, in which each node has at most finitely many children, the value in each node is an aggregation of values in its child nodes, and the hierarchy reflects the partial order on H. We proceed inductively. Let T be any partially constructed tree and let $\beta'_{x'}(k')$ by a leaf node in it such that β' isn't a source. Equation (1) yields $\beta_{x'}(k') = \bigotimes_{y \in \mu_{x'}(k')}\ \alpha_y(\sigma_{x'}(k', y))$, so we create one child for every $y \in \mu_x(k)$, and place value $\alpha_y(\sigma_x(k, y))$ in it. Indeed, the parent is an aggregate of its children, by definition we have $|\mu_x(k)| < \infty$, and since H is a network, we can assume $\alpha_y < \beta'$. We repeat this for all nodes. If this were to go on forever, then by König's lemma, there would be an infinite descending path in the tree, which would yield an infinite descending chain of flows, and this is impossible since the order on H is well-founded. Knowing that the tree is finite and all leaves are sources, by associativity of $\otimes$ we can represent $\beta_x(k)$ as a finite aggregation of values in sources. This concludes the first part of the proof.

Now, take any pair of values $\beta_x(k), \beta_{x'}(k')$, appearing in messages m, m', and let t be the later of the times at which m, m' appear. Let H_t be a network obtained by truncating every flow in H at time t. Now, since aggregation is always

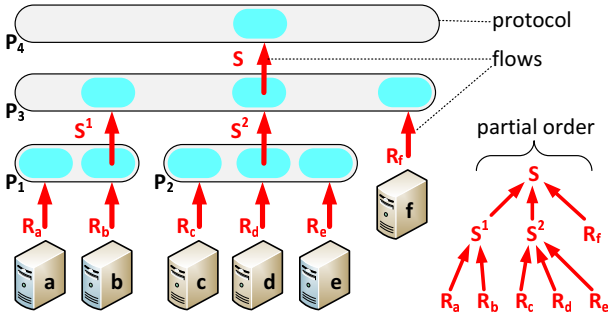

Figure 4. A tree of aggregation protocols and the hierarchy of internal flows between them.

performed on past values, H_t remains well-defined, and all our assumptions still hold. Only finitely many aggregations could happen in a finite time because we have assumed that $\mathcal{T}$ is isomorphic with $\mathbb{N}$. Each involves finitely many nodes. For each of those aggregations, we can construct a finite tree as shown above, and further truncate H_t, to leave only those flows we encountered in the construction of those trees. The resulting network H'_t is finite, so finally, we can apply Theorem 2.1 inductively, starting from the sources, and working towards β, and eventually, we obtain $\beta_x(k) \leq \beta_{x'}(k')$. ■

The practical significance of this theorem is as follows. Suppose that small groups of nodes run internal aggregation protocols. For example, machines a and b run protocol P_1 to aggregate their local values R_a and R_b (Figure 4) and nodes c, d, and e run another protocol P_2 to aggregate R_c, R_d, and R_e. The aggregate values generated by protocols P_1 and P_2 form internal flows S^1 and S^2, respectively. A higher-level protocol P_3 aggregates those values, plus values from node R_f, to produce flow S, consumed by some component P_4. The hierarchy may, of course, be deeper, and since this is a dynamic system in which nodes may join, leave, or fail, the set of nodes running each protocol may change, and internal protocols can be started and terminated as the system grows and shrinks. Theorem 2.2 ensures that as long as protocols are well-ordered, and each of them satisfies the assumptions of Theorem 2.1, flow S is strongly monotonic despite churn, and with no global coordination needed between protocols. This justifies the architecture we propose in Section 3.1, in which each protocol is controlled by a separate membership service, and is managed independently of other protocols.

3. Architecture

3.1. Hierarchical Delegation

In this section we present a practical approach to creating protocol hierarchies such as those shown on Figure 4.

We begin by discussing the internal structure of the client protocol stack. The stack includes three components shown on Figure 1, including the *working component* representing lower layers and the *data flow component* P implementing the core logic. Interaction with P is done via messages that contain values tagged with version numbers (Figure 5).

Initially, P contains no actual protocol logic, and doesn't know what to do with the values it receives; it only contains a *bootstrap code* for contacting a *delegation authority* (DA). Upon request, the DA returns a serialized description of P's stack. It hands out exactly the same code to all of its clients.

The are two classes of DAs: the *root authority* (RA), and all the rest. The RA returns a *root code* that doesn't involve any interaction with other nodes; it simply consumes values, performs internal computations, and sends results back on the same node on which it is running. This code implements the *decision* logic. It runs at a single node in the system at a time (except for brief periods during reconfiguration).

A regular, non-root DA returns *aggregation* code that implements a token ring protocol running among all clients boostrapped from this DA (Figure 6). The aggregation component described in Section 3.2 uses the token ring to aggregate and disseminate values in this local client group; it corresponds to a single protocol in the hierarchy on Figure 4. The group uses a private, local membership service (MS) to self-organize. A single DA manages only a small subset of clients, so the local MS shouldn't experience a heavy load. The code for contacting a local MS, including addresses and all parameters, is embedded in the code returned by the DA.

P's aggregation stack includes a (recursively embedded) data flow component P'. Normally, P' remains inactive. It can prefetch code from its own DA, but doesn't activate the downloaded code, and P doesn't attempt to interact with it. P' stays dormant until the local node becomes the leader of the token ring, at which point it boostraps itself and starts to communicate with P. Once the local node ceases to be the leader, P' is deactivated and all its runtime state is disposed.

The above pattern can repeat recursively: P' contains an embedded P'', which contains P''', and so on (Figure 7). If the node happens to be a leader in all rings it is part of, this recursion terminates with the inner-most component boostrapped from RA. Otherwise, the inner-most P_k is running aggregation code, while the embedded $P_{k'}$ stays dormant.

Delegation authorities form a hierarchy: each DA except for RA has a *parent* DA'. In the code returned by a DA, the embedded component is configured to bootstrap from DA'. As a result, a hierarchy of token rings emerges (Figure 8).

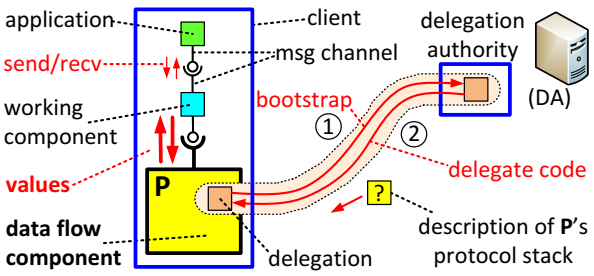

Figure 5. The structure of the client's protocol stack: P's code is bootstrapped from the DA.

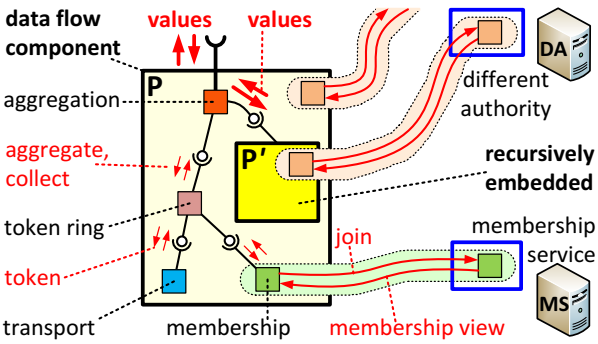

Figure 6. The internal structure of a data flow component P running the aggregation code.

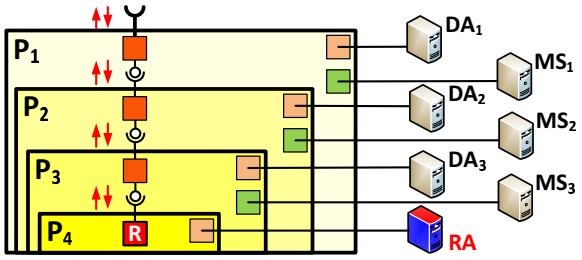

Figure 7. A stack of aggregation components.

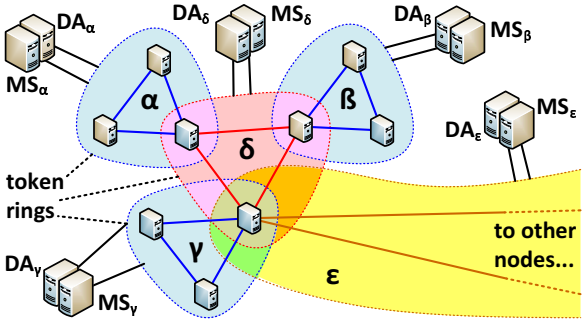

Figure 8. A hierarchy of token rings managed independently by their regional DAs and MSs.

Each data flow component in the protocol stack, and each ring in the hierarchy, is independently bootstrapped from its own DA and independently managed by the associated MS. The only form of cross-layer interaction is, when a data flow component P_k on a client activates, disposes, or exchanges values with the component $P_{k'}$ recursively embedded in it. Different MSs and DAs never interact with one another.

The hierarchy of DAs emerges via the following process. First, the RA is created, and configured to return root code. A single top-level DA is also created with its associated MS; the aggregation code P it returns is configured to bootstrap its embedded P' from RA. All nodes bootstrapped from the top-level DA become members of the top-level ring and one of them always runs the root code. This lays the foundation. The process now continues inductively, by passing around *invitations* (the first invitation created by our top-level DA).

An *invitation* is a small packet containing three elements: a serialized description of a working component (Figure 5), the list of all aggregation rules (specifications for the component named "aggregation" on Figure 6), and the bootstrap code for the DA that issued the invitation. Invitations can be passed around through any channel, for example by email.

An invitation can be consumed directly by a client, by assembling its parts into a protocol stack (Figure 5, Figure 6). Alternatively, the invitation can be used to setup a new DA, with the DA that issued the original invitation as its parent. The new DA can now issue its own invitations, by replacing the bootstrap code in the parent invitation with its own.

The process of passing invitations and setting up the hierarchy of DAs could be performed manually, by administrators, similarly to how one manually sets up the hierarchy of DNS servers. It could potentially also be automated, with the DAs detecting one another via gossip and using peer-to-peer techniques to form hierarchies. The discussion of such techniques is beyond the scope of this paper. However, note that our model, due to its decentralized nature, places very few requirements, and is especially easy to support by such adaptive solutions, for in the light of Theorem 2.2, it suffices that DAs form a tree and never change their parents.

3.2. Aggregation Component

Aggregation components (Figure 7) interact using *value buckets*; one bucket for each input or output flow (Figure 9). When a value arrives from a component higher or lower in the hierarchy, it goes into an input bucket, and when a value in any output bucket changes, it is sent out. Internally, value changes trigger *rules* that update other buckets. All components except the root run *regular* rules, and the lowest-level ones additionally run *client* rules. The root runs *root* rules.

Due to the limited space, in the remainder of this section we discuss only rules for a monotonic, guarded aggregation. Other types of rules are implemented in a similar way.

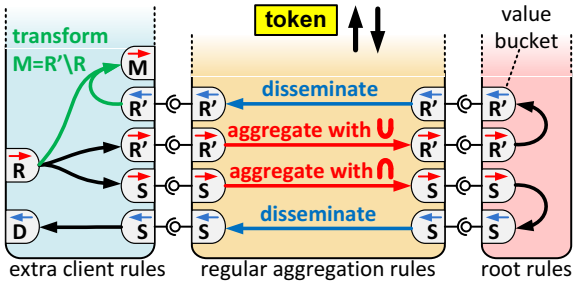

Figure 9. Example aggregation rules (partial)

Values are aggregated by passing tokens around the ring. The ring leader puts a value from its input bucket in a token, and tags it with version $k = (i, j)$, where i is the number of the current membership view, and j is the number of the current aggregation round in the view. Then, each node the token passes by replaces value v in the token with $(v \otimes v')$, where v' is the value from its input bucket. When the token returns to the leader, the aggregated value in it is placed in an output bucket, and in the next round, it is disseminated around the ring, and placed in output buckets of other nodes.

The behavior just described oversimplifies: it covers the case of a *regular* member. A new node starts as a *candidate*. It can read values from tokens, or participate in non-guarded aggregations; by doing so it can catch up with others (obtain state transfer, participate in loss recovery, etc.). To become *regular*, a candidate must do the following (except when all members of the view are candidates, and are automatically promoted; details of the recovery phase omitted for brevity).

Whenever a token passes through a candidate, carrying a partial result v of the current and some, even a partial result v' of the preceding aggregation, the candidate tests whether $v' \leq v''$ holds, where v'' is the candidate's value. If it does, the candidate can replace v with $(v \otimes v'')$, but it doesn't yet become a regular member. Instead, it records version k of the current aggregation, and waits for the next round. Only after a new token arrives with the result of this aggregation (k), the candidate promotes itself to the regular status. If the node later finds out that it's been dropped from the view, it degrades itself to the candidate status. The process of promoting and degrading is done locally, and does not require any kind of coordination with other nodes or with the MS.

The above protocol ensures that aggregation is guarded; a node does not participate in it until it learns at least partial results of the immediately preceding round and ensures that the guarding condition holds. Once a node finds out that its local value affected the result, this is no longer needed.

As noted in Section 2.4, this aggregation is in-order because it is done in rounds, and values placed in the buckets are always those with the highest versions ever received. If the aggregation operator is strongly monotonic, then monotonicity of the aggregated flows follows from Theorem 2.1.

4. Performance

As noted earlier, for reasons of brevity, the scope of this section is limited; we focus on what we believe are two most critical factors affecting the performance of our system: the *latency* of monotonic aggregation in the presence of churn, and the *space overhead* of value representation. To measure the significance of these factors in their purest form, undisturbed by performance of other mechanisms, such as packet forwarding or state transfer, we use simplified protocols.

To evaluate truly large scale scenarios, we had to resort to a discrete event simulation, but to make our results as realistic as possible, only the transport and membership layers were simulated; clients still communicate via asynchronous messages, establish connections, form rings based on membership updates, and serialize transmitted packets. Average network latency is 10ms, and the rings circulate 10 tokens/s.

4.1. Aggregation in the Presence of Churn

In the first experiment, we cause clients to synchronously enter subsequent phases of processing. The integer-valued input flow L informs the protocol of the latest phases $L_x(k)$ entered by each client, and the output flow N instructs each client which phase $N_x(k)$ to execute next. The protocol can be concisely written as $L' = \min L;\ N = L' + 1$. Monotonic aggregation L' computes the last phase entered by the slowest client. After incrementing, this is the last phase that anyone else is permitted to enter. Clients enter their phases instantly, but they do so at different times due to asynchrony and churn. We measure the mean interval between entering subsequent phases as a function of system size and churn. All clients fail and reboot with exponential distribution; the average time to failure (MTTF) is a parameter, and the mean time to reboot is 5s. Rebooted clients are delayed (aggregation is guarded). The token ring size is 8 nodes on average.

The results on Figure 10 show that latency grows as a logarithm of system size (n). It takes about 4 additional token rounds for each layer in the hierarchy (2 rounds each way), for a wide range of churn rates. Even under extreme churn (MTTF=10s), latency grows by a mere 20%; this is because aggregation in different parts of the system is done in parallel, unaffected rings still make progress, and delays caused by membership changes are averaged out across the system. It is worth noting that with 32K nodes and MTTF=10s, the system undergoes about 4K membership changes a second; in such scenarios, approaches based on global membership would suffer from excessive reconfiguration. In our system, reconfiguration after membership change normally takes 2-3 rounds, but each failure disrupts on average $O(1)$, and in the worst case $O(\log n)$ rings. The benefits of hierarchically decomposing the GMS into multiple MSs are thus evident.

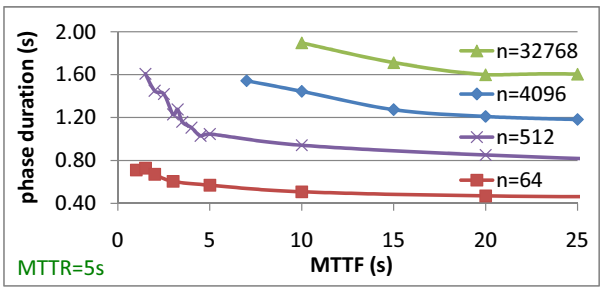

Figure 10. Phase duration as a function of system size and mean time to failure (MTTF).

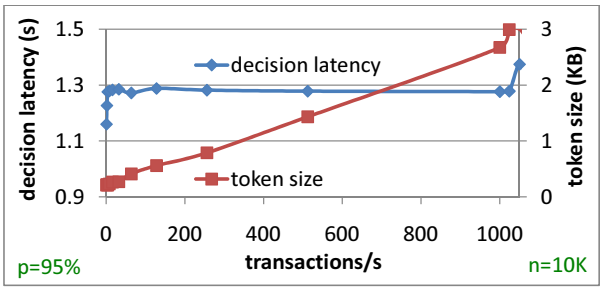

Figure 11. Decision latency and token size as functions of the application event rate (TPS).

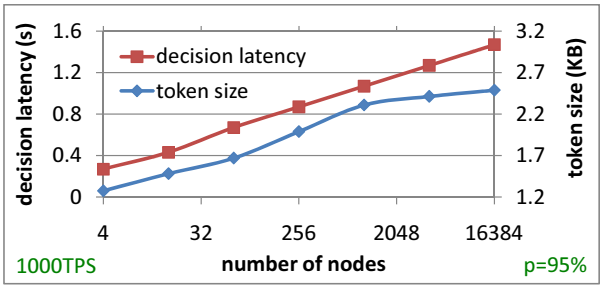

Figure 12. Decision latency and token size as functions of system size (1000 events/s).

4.2. The Overhead of Value Representation

In the preceding experiment, all values would fit in a constant amount of space. In many real protocols, this is not so; values could occupy much space in the tokens, and to bound resource usage, we have to limit token sizes, truncating values that cannot fit. As a result, smaller batches of events can be handled in parallel, and the system slows down.

To illustrate this, in the second experiment we run a simplified commit protocol: each client receives transactions at a fixed rate, and independently decides to commit or abort,

with probability adjusted so that a fraction p of transactions commit globally. Values in input flows C, A are sets of identifiers of transactions that individual nodes wants to commit ($C_x(k)$) or abort ($A_x(k)$). Output flows C', A' carry global decisions. An internal flow D records identifiers of transactions for which decisions have been made. The protocol can be written as $C' = \bigcap(C \backslash D);\ A' = \bigcup(A \backslash D); D = C' \cup A'$. Aggregations C' and A' are guarded and monotonic.

Each value, as a set of numeric transaction identifiers, is encoded as a tuple $((a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k), c)$. Each pair (a_i, b_i) represents a set $\{a_i, a_i + 1, \ldots, b_i\}$. The number k of these pairs is limited by a parameter $k_{max} = 100$. The interpretation is as follows: for every $i \leq c$, element i is in the set iff it is within any of the ranges (a_i, b_i), whereas for $i > c$, this is undefined. Operators $\cup$ and $\cap$ are modified accordingly to correctly operate on such "truncated" sets. If multiple such values are combined using $\cup$ or $\cap$, information is often lost in the process because some of the ranges (a_k, b_k) don't fit within the limit k_{max} and c may become lower. Because of this, a single aggregation may no longer suffice to propagate all information from clients to the root.

In the first scenario in this experiment, we fix the commit probability at $p = 95\%$ in a group of $n = 10000$ nodes, and vary the transaction rate, measuring the time until the slowest client commits or aborts (Figure 11). As expected, token size grows linearly: the number of numeric ranges (a_i, b_i) is proportional to the number of events to report in each round. Latency is virtually unaffected. Processing each token takes $\approx$200μs (on Pentium 4, 3.8 GHz); 75% of that is the cost of serialization. As tokens grow, more CPU is needed, but not extra rounds. Only when the event rate exceeds $\approx$1050 TPS, k_{max} is reached, values are truncated aggressively, transactions pile up, and latency shoots to infinity (not shown).

In the second scenario, the rate is fixed at 1000 TPS, still $p = 95\%$, and we vary the system size (Figure 12). Latency and token size grow only logarithmically, and the latency is nearly the same as in the preceding experiment. Again, we find that as long as the average value that's being aggregated remains beneath the k_{max} threshold, the system responds to the increased load by increasing the token sizes, and latency remains virtually unaffected. At $\approx$32K nodes we're starting to approach k_{max}, and the system becomes saturated; if we scale further, transactions start piling up. The system, however, does not collapse; it keeps aggregating at a steady rate.

In the last scenario, we relax token size, $k_{max} = \infty$, and we vary p with other parameters constant, to find how much data would otherwise be truncated (Figure 13). We find that when transactions commit at random ($p = 50\%$), values can occupy up to 12 KB/token; with 10 tokens/s, this means $\approx$1 Mbps per-node control traffic in every ring, so the overhead can be fairly substantial, and truncating is necessary. In real systems, each ring could adjust its own token rate and k_{max} adaptively, based on the measured latency and bandwidth.

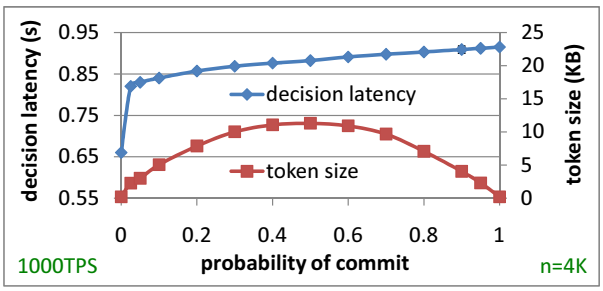

Figure 13. Latency and space overhead when aggregated data is not truncated ($k_{max} = \infty$), with 4096 clients and 1000 transactions/s.

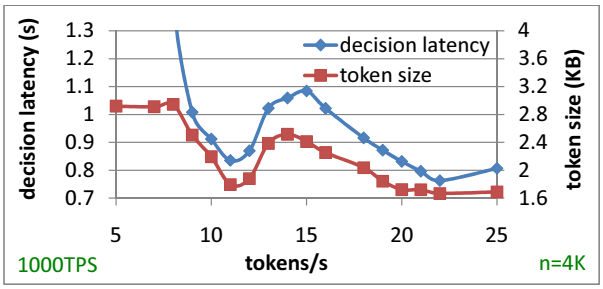

Figure 14. Varying the token circulation rate.

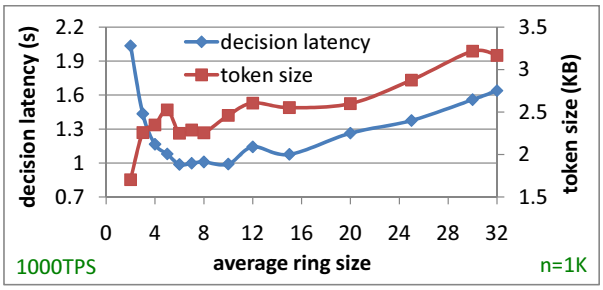

Figure 15. Varying average ring size (fanout).

4.3. Hierarchy Depth and Aggregation Rate

To conclude, we look at the effects of varying token rates (Figure 14) and ring size (Figure 15). Having several tokens chasing each other (e.g. >12 tokens/s in an 8-node ring with $\approx$10ms latency) results in redundant work. Wrong ring sizes also hurt latency, for either hierarchy is deep, or it takes long to aggregate in each ring. A bad choice of parameters can affect performance by a factor of 2. Replacing rings with trees may partially alleviate the issue. In practice, ensuring that the overall hierarchy is balanced appears to be a bigger challenge. Although our architecture is flexible, it doesn't allow protocols to change parents in the hierarchy, making algorithms for self-balancing trees harder to apply.

5. Related Work

Our work is closely related to, and was strongly inspired by the rich prior literature on I/O automata (IOA) [17]. IOA pioneered an approach, in which distributed protocols are modeled as components that operate on and transform event streams. It's been successfully used to specify a number of protocols [16], and reason about composition [11], also in real systems such as Ensemble [9]. TLA [13] is another major model in this space; it has also been explored in the context of composition [1], and used to formalize Paxos [14], but we're not aware of prior work on applying TLA to flows.

Whereas IOA has focused on the compositional structure of protocols within individual endpoints, our work retains a similar functional flavor, but with a focus on flows. By eliminating the node-centric aspects of IOA, we gain flexibility that can be exploited to create freedoms: freedom to create a hierarchy independently from the way the protocol aggregates and disseminates, to batch events and exchange information in ways convenient to the runtime system. Although this paper does not focus on implementation, systems based on our model can use these freedoms to achieve scalability and to adapt to the properties of their runtime environments.

High-level specifications similar to our dataflow notation have been used for workflow modeling [10] and web service choreography [2], but generally, specifications derived from process calculi are too weak to express strong properties [8].

Work on declarative networking [15] shares some of our goals, such as support for concise, high-level protocol specifications. However, their architecture, unlike ours, has not been designed from ground up to support hierarchical, scalable protocols with strong reliability properties, and we are not aware of any attempts to use their work in this context. The mechanisms they use are also very different from ours.

There has been much research on data flows in areas such as VLSI or DBMS, but also publish-subscribe [6] or routing [18]; the advantages of asynchronous, parallel and pipelined processing are well understood. Flows encountered in those systems, however, aren't *distributed* in the same sense as in our work; they are sequences of events, and transformations on them are performed locally. We're not aware of any prior attempts to combine data flow processing with IOA style of modular specifications in a manner similar to our approach.

There has been much prior work on aggregation in sensor networks, even with stronger properties [3], but the kinds of properties targeted by those systems revolve mostly around security, and we are not aware of any example uses of these techniques in protocols such as reliable multicast.

Much research focused on making GMS scalable, in particular also through the use of hierarchy [12], but scalability in traditional GMS-driven protocols, such as virtual synchrony, is ultimately limited by the fact that each member of the group must ultimately receive the complete global view.

6. Conclusions

We proposed a new approach to building distributed protocols with strong properties that does not rely on GMS, and that combines ideas from areas such as IOA, data flows, and sensor networks. We developed a theory to reason about our model, a supporting architecture, and we briefly reported on the performance of our initial prototype. Our approach appears to be fairly general, scalable, and very churn-tolerant.

7. Acknowledgements

This work was supported in part by grants from AFOSR, AFRL, NSF, and Intel Corporation. We'd like to thank Robbert van Renesse and Daniel Freedman for their comments.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *TOPLAS*, 1995.
[2] A. Banerji et al. Web Services Conversation Language (WSCL). *http://www.w3.org/TR/wscl10/*.
[3] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. *CCS*, 2006.
[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *JACM*, 1996.
[5] G. Chockler, I. Keidar, and W. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computer Surveys, 33(4):1, pp. 43, Dec 2001.*, 2001.
[6] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive pub/sub systems. *EDBT'06*.
[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.
[8] R. Fuzzati and U. Nestmann. Much ado about nothing? *http://www.brics.dk/NS/05/3/*, 1995.
[9] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. *TACAS*, 1999.
[10] IBM et al. Business process execution language for web services (BPEL). *http://www.ibm.com/developerworks/*, 2007.
[11] B. Jonsson. Compositional specification and verification of distributed systems. *TOPLAS*, 1994.
[12] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for wans. *TOCS*, 2002.
[13] L. Lamport. The temporal logic of actions. *TOPLAS*, 1994.
[14] L. Lamport. The Part-Time Parliament. *TOCS*, 1998.
[15] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SOSP*, 2005.
[16] N. Lynch. Distributed algorithms. *Morgan Kaufmann*, 1996.
[17] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *PODC*, 1987.
[18] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SOSP*, 1999.
[19] K. Ostrowski, K. Birman, D. Dolev, and J. Ahnn. Quicksilver Live Objects. *http://liveobjects.cs.cornell.edu/*, 2008.
[20] F. Schneider. Byzantine generals in action: implementing fail-stop processors. *TOCS*, 1984.