Implementing Reliable Event Streams in Large Systems via Distributed Data Flows and Recursive Delegation

Krzysztof Ostrowski[†], Ken Birman[†], Danny Dolev[§], and Chuck Sakoda[†]
[†]Cornell University
Ithaca, NY 14853, 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 and virtual synchrony, are hard to implement in a scalable manner. They are typically expressed in terms of global membership views. However, global membership is non-scalable. We propose a new 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 events 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 service while still supporting stronger properties. This paper focuses on the theoretical model and its base properties; in particular, on the concept of monotonic aggregation. We present a high-level architecture overview and initial performance results.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Performance, Reliability, Theory

Keywords

Aggregation, Data Flow, Membership, Protocol, Scalability

1. INTRODUCTION

We believe that there is a need for eventing middleware that can support dissemination on a massive scale while providing strong reliability guarantees. For example, in data centers, reliable multicast or publish-subscribe groups spanning over thousands of machines could store dynamic configuration state, such as partitioning of resources across applications. Scalable mechanisms of this sort could also be used to *consistently* and *reliably* disseminate security policy updates, key revocation requests, or software patches, thus enabling fast and well-coordinated responses to threats. Unfortunately, existing multicast and publish-subscribe technologies force developers to choose between scalability and strong reliability properties. Group communication platforms that implement virtual synchrony and other types of strong guarantees are far less scalable than peer-to-peer content dissemination networks, whereas the latter are often unsuitable for the types of applications that require stronger forms of consistency. In this paper, we propose a new approach to implementing reliability that can simultaneously achieve both goals.

Our work is focused on distributed replication protocols, in which nodes can dynamically join, leave, and fail by crashing (and rejoin with new identity), and where churn can be high. High churn is typical of peer-to-peer scenarios, but it can also happen in data centers during load surges: timing-out connections can easily be mistaken for failures. Churn tolerance is the key to stability in such systems because an inadequate response, unnecessary reconfigurations and retransmissions could easily worsen the situation, lead to cascading effects, propagate across the network, and cause massive outages.

The term *strong properties* in this paper refers to quasi-absolute [3, 7] properties such as virtual synchrony, atomic broadcast, commit, transactions, state machine replication, or consensus with dynamic membership. In most existing implementations of protocols that offer strong properties, protocol participants are controlled by sequences of membership views generated by a *global membership service* (GMS), which could be external, or a part of the protocol itself. Strong guarantees are expressed in terms of the global views, which, as explained below, is a major factor that limits scalability. Our work would not be applicable to gossip protocols, for example. These avoid global views, but whereas we aim at strong properties, they usually are limited to weaker (convergent) ones. The scenarios and challenges these systems address differ significantly from ours.

GMS-mediated reliability is well-understood and frequently used in practice, but it has limitations. As the system grows in size, the frequency of membership changes increases as $\mathcal{O}(n)$ of the system size (n). Eventually, this can become a serious burden on the members, and on the GMS itself. Second, in protocols such as virtual synchrony each membership change triggers $\mathcal{O}(n)$ work. With the frequency of membership changes and the cost per change both at the level of $\mathcal{O}(n)$, this can lead to $\mathcal{O}(n^2)$ cost that rapidly becomes prohibitive. Batching changes is an obvious option to consider, but it can result in other problems: in many protocols, progress cannot occur if a member becomes unresponsive until the GMS excludes it from the view, and batching can delay the GMS's reaction. These problems compel a rethinking of the relationship between membership and reliability, and suggest that globally visible and consistent membership should not be a part of large-scale reliability models.

We propose a novel approach, in which the centralized GMS is replaced with a large number of (*local*) *membership services* (MS), organized into a hierarchy. In our approach, a single MS manages only $O(1)$ nodes, and if the hierarchy is balanced, each node interacts with only $O(1)$ other nodes and MSs on average, and $O(\log n)$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'09, July 6-9, Nashville, TN, USA.
Copyright 2009 ACM 978-1-60558-665-6/09/07... $10.00.

in the worst case. No part of the system thus becomes a bottleneck. Furthermore, churn and membership changes have limited impact. While a small portion of the nodes is undergoing reconf guration, other parts of the system continue to independently make progress. Often, reconf guration can happen in parallel, and the cost of churn can be amortized. In our simulations, even with 32000+ nodes and the average time to a node failure (MTTF) at the order of just 10-20 seconds, key performance metric drops by only 20%. Of course, in practice the effectiveness of our approach might vary; if all nodes share the same hardware infrastructure and compete for resources, failures and load surges in different parts of the network might be correlated, and even with our approach, performance could degrade more sharply than what is shown in our simulations. Nevertheless, by decentralizing the handling of membership and removing bottlenecks at the protocol level, our approach can alleviate the problem. We do not claim that our techniques are applicable universally; only that they offer scalability advantage over GMS-centric approaches.

The key idea is illustrated on Figure 1. Nodes participating in the protocol form a hierarchy of token rings. First, the entire system is partitioned into small rings (α, β, and γ on Figure 1), each node in exactly one ring. Then, these lowest-level rings are again clustered into small groups, and within each group of rings, selected leader nodes form a higher-level ring (δ on Figure 1). The clustering continues recursively, with higher-level rings (ϵ on Figure 1), up to the *root*-level ring that connects all parts of the system together. Each ring, at any level of this hierarchy, is independently controlled by a private pair of local services: a local *delegation authority* (DA) and a local *membership service* (MS). Each DA/MS combo is responsible for controlling only its own private ring, and it does not need to interact with other DAs/MSs. There is no need for a GMS; global membership never materializes in any part of the system. The local membership views, nevertheless, form a hierarchy that evolves in a well-controlled manner. At f rst, this decentralized structure may seem to be prone to chaotic behavior; however, in this paper we formally prove that in fact, it is able to implement strong properties.

To make use of a hierarchical and decentralized membership, the process of achieving strong properties also needs to be hierarchical. At a high-level, the idea resembles the approach used in the RMTP [28] multicast protocol. In RMTP, groups of recipients also form a hierarchy. At each level in the hierarchy, recipients locally cooperate on recovering the lost packets, forward data to one-another, and report aggregate status to the higher levels. This way, loss recovery can occur in parallel in multiple places in the network, each member has a limited number of peers, and nodes at higher levels in the hierarchy deal mostly with aggregate status. However, guarantees offered by RMTP are only best-effort. Our approach extends and generalizes these techniques to protocols with stronger guarantees.

Our approach is based on the novel concepts of *distributed flow* (DF) and *monotonic aggregation* (MA). A distributed f ow captures the state and progress of the protocol in a decentralized fashion, and monotonic aggregation is a tool that allows this decentralized state and progress to be composed reliably and hierarchically.

The process is illustrated on Figure 2. In each bottom-level ring (α, β, and γ), members locally cooperate, calculate their aggregate status, and report it to the higher layers of the protocol in the form of distributed f ows (f ows α, β, and γ on Figure 2). The difference with respect to RMTP-like protocols is that reporting at each layer is performed in a manner coordinated by the local MS. This allows each layer to provide certain guarantees locally. For example, if the state being aggregated includes information on which packets are *stable*, i.e., received by all nodes in a certain portion of the network, each layer can guarantee that once it reports a certain packet as stable, it will deliver on the promise: the packet will either remain

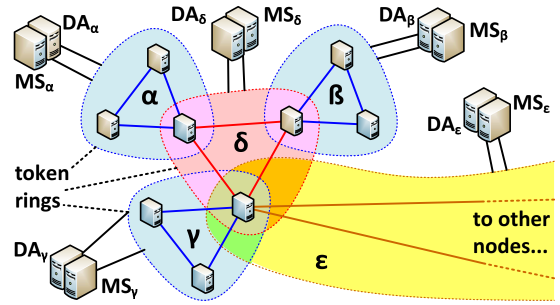

Figure 1: In our approach, nodes are organized into a hierarchy of token rings, each ring of size $O(1)$, and managed by its own, private pair of local services: a local *delegation authority* (DA), and a local *membership service* (MS).

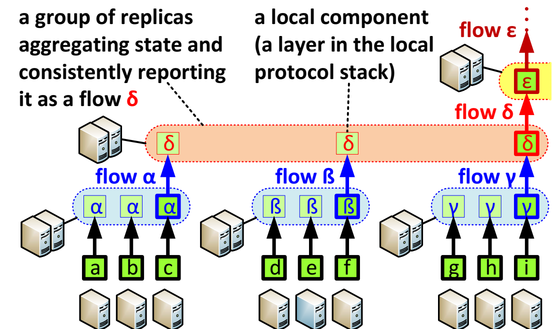

Figure 2: One each node, the protocol stack involves a number of components at different layers; each of these components is a member of a single token ring, and together with components on other nodes, it calculates an aggregate state. The aggregated state is consistently reported to components at higher layers in the hierarchy. We refer to these exchanges of aggregated state between different protocol layers as *distributed data flows* (DF). The local MSs help to ensure that DFs have strong properties.

stable, and always be consistently reported as such in the future, or the layer will separate itself from the protocol and rejoin (with new identity). The ability for a DF at each layer to *remember* information and make *commitments* is expressed as *monotonicity*, a strong property of f ows that can be implemented thanks to the local MSs. Monotonic aggregation is a mechanism that allows this property to be built up hierarchically: f rst, all bottom-level rings (α, β, and γ) achieve monotonicity thanks to their local MSs (MS_α, MS_β, and MS_γ), then higher-level rings (δ) monotonically aggregate f ows coming from the bottom layers, and so this continues up to the root.

Membership changes are viewed as perturbations that disrupt the integrity of the f ows. When a member of a token ring crashes, one of the replicas that held some aggregated state that may have been reported to higher layers is now lost. If the way state is aggregated and reported is not done properly, this could lead to inconsistency. Our f ows are designed (and formally proven) to tolerate perturbations by locally repairing rings and re-generating lost information.

Our method appears to be fairly general and applicable to modeling a variety of protocols; indeed, we have designed a programming language, in which one can express the semantic of protocols in a high-level, declarative fashion, as data f ow dependencies, and then compile it down into scalable executable code [25]. We are nearing the completion of a compiler framework that will be released as a part of our existing *Live Distributed Objects* (LO) platform [1, 27].

This paper could have focused on a number of different aspects

of our approach: the theoretical foundations, the protocol modeling language, the protocols and system architecture, and performance evaluation. We decided to focus primarily on the theoretical model and its properties, for they lay the foundations on which our system is built. Our specif c architectural decisions and protocols, brief y outlined in Section 3, follow almost automatically as direct consequences of the theorems we prove in Section 2. Many of the details, e.g., how to build token rings, are borrowed from our prior work on the QSM scalable multicast engine [26]; we omit them for brevity. For the same reason, comprehensive performance evaluation of our approach is outside the scope of this paper. However, we do brief y report on the initial simulation results to give the reader some intuition about the two major types of overheads our approach incurs. Finally, we present only those def nitions, theorems, and proofs that are essential to understanding our approach. In particular, we focus only on the correctness of the constructed protocols; a discussion of liveness requires much more lengthy treatment, and will be presented elsewhere. Our initial experiments suggest that in practice, our system achieves steady, uninterupted progress even in very large conf gurations and at high churn rates; we never observed a single instance of deadlock (or livelock) in any of our experiments.

This paper makes the following contributions.

- It proposes a new concept of a *distributed data flow*, a stream of events distributed across the network, and a model that allows the global behavior of distributed multi-party protocol to be expressed in a purely functional style, as graphs of distributed functions that operate on and transform such f ows.
- It introduces basic classes of purely functional operations on f ows: *dissemination*, *aggregation*, *transformation*, and *distribution*, and illustrates their practical use by dissecting and analyzing parts of a simple reliable multicast protocol.
- It explains how strong reliability properties map to the properties of these basic building blocks; in particular, it explains how *strong monotonicity*, a core concept in our model, can be used to reliably record and recall distributed protocol state.
- It introduces and proves two theorems characterizing the conditions under which strong monotonicity can be achieved using simpler properties, and how it can be hierarchically composed. The theorems capture the essential properties that the protocol and runtime architecture must satisfy to be correct.
- It brief y outlines an architecture that allows f ow hierarchies to be created through recursive delegation, and employs multiple membership services to support a large group of clients.
- It shows how core protocol semantics can be cleanly decoupled from the construction and maintenance of the underlying hierarchy; a new kind of f exibility unseen in prior work.
- It reports on the early performance evaluation with a real protocol stack running in a discrete event simulator, focusing on two major types of overheads our model incurs. The results show that our system can handle group sizes and churn rates that would pose a major challenge to GMS-based techniques.

2. MODEL

2.1 Distributed Flows

We def ne a *distributed data flow* (or simply a *flow* or DF) as a set of events exchanged between two layers in the protocol stack, and across some set of machines in the network. This is illustrated on Figure 3. Suppose that the protocol stack has two layers, A and B. On each physical machine x_i that participates in the execution of this protocol, there are two software components A_i and B_i that

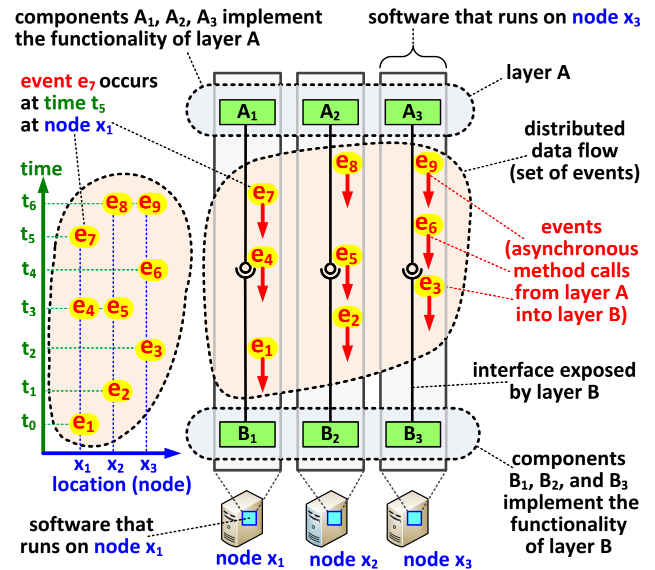

Figure 3: A *distributed data flow* (DF) is defined as a set of events exchanged between two protocol layers. Each DF is distributed in space (the events can appear on different nodes), and in time (events keep flowing over time). Each event e_i in a flow is modeled as a quadruple of the form $e_i = (x_i, t_i, k_i, v_i)$. For example, $e_7 = (x_1, t_5, k, v)$ for some version $k \in \mathcal{K}$ and value $v \in \mathcal{V}$.

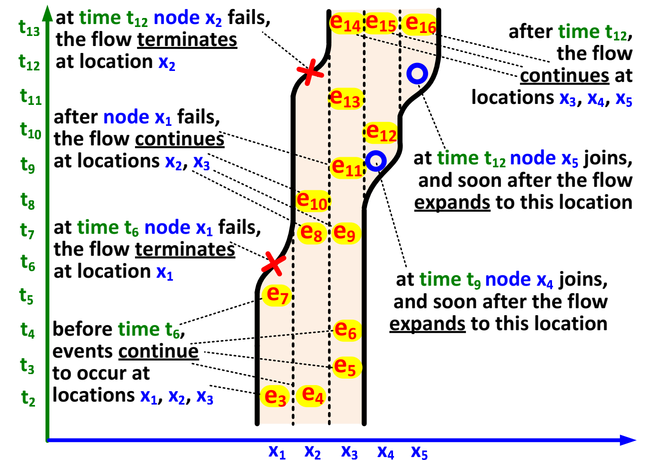

Figure 4: The set of locations at which events appear in a flow can change as nodes join, leave, and fail. One can think of the flow as terminating at some locations and expanding onto new ones. Throughout an infinite history of the system, the flow can appear on infinitely many nodes, but at any given point in time, events continue to appear only at a finite subset of these nodes.

belong to these layers and locally implement the respective functionality. Pairs of components A_i and B_i are connected through their APIs; suppose that A_i is calling methods of B_i. Each time A_i invokes a method, we model this as an event f owing from A_i to B_i. For simplicity, we assume all calls to be asynchronous (if they are not, one could model the replies as events f owing backwards). The set of all events of the same type (for example, representing invocations of the same method), f owing in the same direction, at any time, and between every pair of components A_i and B_i (for all the different i), constitutes what we call a distributed data f ow.

Note that events in a data f ow are distributed in space, since they

appear on the different nodes x_i participating on the protocol, and in time, since pairs of components on the same node typically continue interacting over a period of time; we model each interaction as a separate event. Furthermore, the set of nodes at which events keep appearing is not f xed (Figure 4). As nodes dynamically join, leave, and fail, events start and stop appearing at those nodes. One can think of the data f ow as shrinking (terminating at certain locations) and spreading (expanding onto the new locations) over time.

Formally, each event in a f ow is modeled as a quadruple of the form (x, t, k, v), where $x \in \mathcal{X}$ is the *location* at which the event occurs, $t \in \mathcal{T}$ is the *time* at which this happens, $k \in \mathcal{K}$ is a *version* that the event is tagged with (for now, one might think of versions as sequence numbers; we discuss this later), and $v \in \mathcal{V}$ is the actual *value* (payload) the event is carrying (method arguments, results, or other data encapsulated within the event). The sets of all nodes, times, versions, and values are denoted as $\mathcal{X}$, $\mathcal{T}$, $\mathcal{K}$, and $\mathcal{V}$, respectively. Given event $e = (x, t, k, v)$, we denote the four components of the quadruple as: $\chi(e) = x$, $\tau(e) = t$, $\kappa(e) = k$, and $\nu(e) = v$.

For example, if component A_{lion} on node *lion* invokes a method *foo*(1000) on component B_{lion} at time 10, and it is the f fth such method call on node *lion*, one would express this fact in our model as "$(lion, 10, 5, 1000) \in foo$". By convention, we name the f ows after the methods, the invocations of which they record. Here, f ow *foo* records calls to method *foo* (made from layer A into layer B).

Within the set of nodes $\mathcal{X}$, we further distinguish nodes that are *faulty* and *non-faulty*. The latter are nodes that eventually begin and never cease to execute the protocol; their set is denoted as $\mathcal{X}_0$. We assume the *fail-stop* model [29], i.e., a node $x \in \mathcal{X} \setminus \mathcal{X}_0$ that fails can only reboot with a new identity $x' \in \mathcal{X}$, where $x' \neq x$.

We assume that $\mathcal{T}, \mathcal{K}, \mathcal{V}$ are ordered by $\leq_{\mathcal{T}}$, $\leq_{\mathcal{K}}$, and $\leq_{\mathcal{V}}$ (or $\leq$, for short). To keep it simple, we assume that orders $\leq_{\mathcal{T}}$ and $\leq_{\mathcal{K}}$ are total. We assume that $\mathcal{T}$ represents a global, absolute time, and is isomorphic with the set of real numbers $\mathbb{R}$. The latter assumption is not critical, but it simplif es the presentation. Our results carry over to the more general case. Time is not central in our model, it is not part of event payload, and is not observable by nodes; it is used only for modeling purposes, mostly to root the model in physical reality, and when def ning aspects such as liveness that are beyond the scope of this paper. Most def nitions and results are expressed only in terms of locations $\chi(e)$, versions $\kappa(e)$ and values $\nu(e)$. One technical requirement we make is that only f nitely many events can occur in a f ow up to a given point in time.

As mentioned earlier, for now one can think of versions $\kappa(e)$ as sequence numbers indexing method calls. This will not be the case for all f ows, but it is a convenient interpretation for most f ows. In general, version numbers are required to satisfy two requirements.

First, given f ow α, for any two events $e, e' \in \alpha$ that f ow at the same location, the later one has a higher version. Formally:

$$\chi(e) = \chi(e') \wedge \tau(e) \leq \tau(e') \Rightarrow \kappa(e) \leq \kappa(e') \,, \tag{1}$$

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

Notice the assumption $\chi(e) = \chi(e')$; in general, we cannot assume anything about events e, e' at different locations $\chi(e) \neq \chi(e')$.

Second, if two events e, e' have the same version and f ow at the same location, they must carry the same value; in other words, if we f x a location, then the value is a function of the version. Again, we do not assume anything about events appearing on different nodes. Formally, for any f ow α, and for all $e, e' \in \alpha$, the following holds:

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

For notational convenience, we often use the term $\alpha_x(k)$ to refer to value $\nu(e)$ of any event $e \in \alpha$ such that $\chi(e) = x$ and $\kappa(e) = k$; one can think of $\alpha_x(k)$ as "the k-th value f owing at location x". If no event ever f ows at x with version k, then $\alpha_x(k)$ is undef ned. Equation (3) ensures that if $\alpha_x(k)$ is def ned, it is well-def ned. For example, if $(lion, 10, 5, 1000) \in foo$, then $foo_{lion}(5) = 1000$.

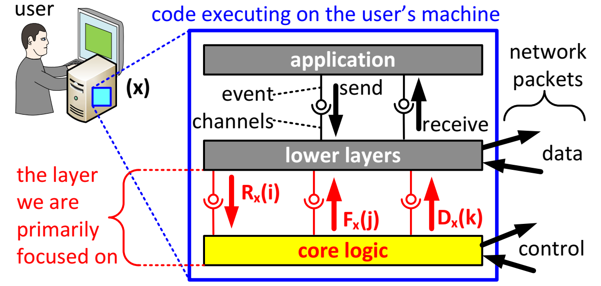

Figure 5: Core logic of reliable atomic multicast as a mapping from an input flow R into output flows F and D. Values $R_x(i)$ in flow R carry sets of identifiers of packets that have been received, e.g., value $R_x(i) = \{1..25,\ 28\}$ would represent a notification that messages with identifiers $1..25$ and 28 arrived on node x. Values $F_y(j)$ in F carry sets of forwarding requests of the form (y, i), where y is the destination, and i the identifier of the packet to forward to y. Values $D_z(k)$ in D carry sets of identifiers of packets that can be delivered to the application.

For certain types of f ows, a much stronger property is satisf ed: version determines the value across all locations. We refer to such f ows as *consistent*. Formally, f ow α is consistent if for all pairs of events $e, e' \in \alpha$ (even at different locations), the following holds:

$$\kappa(e) = \kappa(e') \Rightarrow \nu(e) = \nu(e') \,. \tag{4}$$

If α is consistent and terms $\alpha_x(k), \alpha_{x'}(k)$ are def ned for locations $x, x' \in \mathcal{X}$ and version $k \in \mathcal{K}$, then Equation (4) implies $\alpha_x(k) = \alpha_{x'}(k)$. Thus, with consistent α, one can further abbreviate $\alpha_x(k)$ to simply $\alpha(k)$ and think of the latter as "the k-th value in f ow α".

In our model, consistent f ows often carry decisions or aggregation results (this would be the case for $\alpha, \beta, \gamma, \delta$, and ϵ on Figure 2). Versions in such f ows are no longer sequence numbers assigned on each node individually. To guarantee consistency, we def ne them as tuples that include the number of the membership view and the number of the aggregation round in which the particular decision or aggregated value has been generated. For example, the term $\alpha(k)$ for $k = (5, 3)$ would represent the value aggregated in the 3rd round in the 5th membership view (more details are given in Section 3).

In modeling protocols through data f ows, we focus primarily on *control* f ows, where each value represents a protocol state or a decision, not on the f ows of application data. In every protocol, one can distinguish the part of the stack that implements the protocol's *core logic* that deals with high-level aspects such as deciding when packets are stable, ready to cleanup, or which nodes missed packets and require forwarding, whereas *lower layers* of the stack might be involved in tasks such as physical network transmissions, buffering, or interaction with the application (Figure 5). Here, we focus only on the core logic and its interactions with the rest of the stack.

To support protocols that stream data at high rates, we focus on batched processing, where each value in a f ow can carry decision or state for multiple application events, and the processing at all layers in the protocol is done for sets of such events in parallel. The values carried by events in most f ows we consider will be sets of numeric packet identif ers. We will explain this using an example.

Example. Consider a reliable atomic multicast protocol: a packet that is delivered on any node should be eventually delivered on all non-faulty nodes. As mentioned earlier, interfacing the application, packet caching and forwarding belong to lower layers, whereas the

core logic makes distributed decisions such as what packets to forward or deliver. It does not handle actual data, it "pulls the strings". It can be modeled as a layer that consumes a single distributed flow R, and "transforms it" into two flows F and D, defined as follows.

Flow R carries into the core logic layer information about packets that have been received. Whenever a new packet arrives on node x at time t, we model it as an event from lower layers to core logic on node x, some $e = (x, t, k, v) \in R$, where the value $v = \nu(e)$ is the set of identifiers of all packets received by node x until time t.

For example, if the packet that has just arrived on x has identifier 28, and earlier x received all packets with identifiers from 1 to 25, the value of our event is $v = \nu(e) = \{1..25, 28\}$. Thus, as postulated earlier, each event flowing into the core logic layer can carry information about multiple packets at a time. Since packets keep arriving, the core layer on each node will continue to receive such batched notifications, each reporting a larger set of packet identifiers. In our abbreviated notation, $R_x(k_1) \subseteq R_x(k_2) \subseteq \ldots$, for a certain increasing sequence of versions $k_1 \leq k_2 \leq \cdots \in \mathcal{K}$.

We assumed that the set $R_x(k)$ contains identifiers of *all* packets that node x received up to a point in time, but we do this only for *modeling* purposes. In reality, of course, the lower layers could report the received packets in the incremental fashion. In our methodology, we can treat this as an optimization that can be introduced at a compilation stage (while transforming specifications into code). Modeling $R_x(k)$ as sets of *all* received packet identifiers allows us to express various types of progress in the protocol as a monotonicity on the sequences of values (this is discussed in Section 2.3).

Since packets can arrive in random fashion, the sets of identifiers reported in flow R in general would not be synchronized between nodes, so the sequences of values $R_x(k)$ and $R_{x'}(k')$, for different nodes $x \neq x'$, might be unrelated. Thus, R is not a consistent flow.

Flow D carries delivery decisions out of the core logic. For each event $e \in D$, its value $\nu(e)$ is a set of identifiers of packets that the lower layers (which manage receive buffers and cache packets) can deliver to the application. Here again, core logic does not deal with the actual application data; it only "reasons" at the level of packet identifiers. As with R, set values in D would be locally increasing: for any x, $D_x(k'_1) \subseteq D_x(k'_2) \subseteq \ldots$ for some $k'_1 \leq k'_2 \leq \cdots \in \mathcal{K}$. This is because once some node is permitted to deliver a packet, the decision is irreversible; hence for any x, $D_x(k)$ must grow with k.

Unlike flow R, flow D would in general need to be consistent to achieve strong semantics. We elaborate on this in Section 2.3.

Flow F carries requests to forward packets. Values $F_x(k)$ in this flow are slightly more complex: it is not enough to inform the lower layers which packets will need to be forwarded, it is also necessary to specify the destination. For each event $e \in F$, its value $\nu(e)$ is a set of pairs (y, i), where y is the destination that x should forward the packet to, and i is the identifier of the packet to be forwarded. For example if $(x, t, k, \{(a, 5), (b, 9), (c, 3)\}) \in F$, if means that at time t, the core logic layer at node x has requested that the lower layers forward packet 5 to node a, packet 9 to node b, and packet 3 to node c. In general, flow F will not be consistent: different nodes usually forward different sets of packets to different peers and their forwarding decisions are made in a highly decentralized fashion.

To illustrate the protocol in action, suppose there are 3 nodes: x, y, and z, and that a sequence of 20 packets with identifiers 1..20 arrives from an external source. Nodes x and y receive all 20 packets, but z loses packets 10..15 (Figure 6). The lower layer components on all nodes report this to core logic; we model this as follows:

$$e_1 = (x, t_1, k_1, \{1..20\}) \in R\,, \tag{5}$$
$$e_2 = (y, t_2, k_2, \{1..20\}) \in R\,, \tag{6}$$
$$e_3 = (z, t_3, k_3, \{1..9, 16..20\}) \in R\,, \tag{7}$$

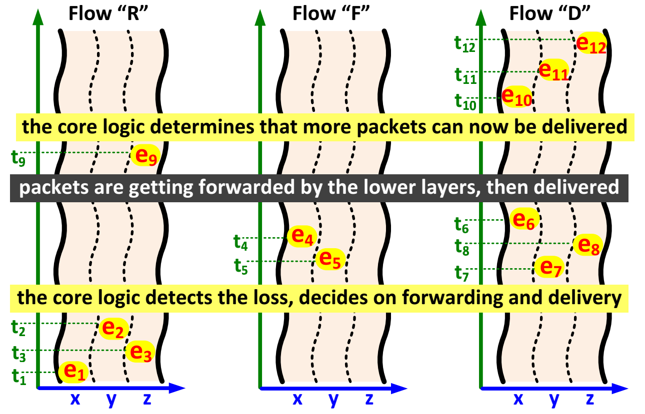

Figure 6: The initial arrival of multicast packets in our example is modeled as events $e_1, e_2, e_3 \in R$. The loss on node z triggers forwarding requests $e_4, e_5 \in F$, and delivery decisions for the subset of packets that are present on every node, $e_6, e_7, e_8 \in D$. The receipt of forwarded packets is reported as $e_9 \in R$. Finally, this triggers delivery of the entire sequence: $e_{10}, e_{11}, e_{12} \in D$.

for some $t_1, t_2, t_3 \in \mathcal{T}$ and some $k_1, k_2, k_3 \in \mathcal{K}$.

Later, after some network communication has taken place, core logic components on nodes x, y, and z detect that node z is missing packets 10..15. Having detected the loss, they decide that x should forward packets 10..13, and that y should forward packets 14..15. The components on x and y then make calls to the lower layer to request the actual forwarding; we model these requests as follows:

$$e_4 = (x, t_4, k_4, \{(z, 10), (z, 11), (z, 12), (z, 13)\}) \in F\,, \tag{8}$$
$$e_5 = (y, t_5, k_5, \{(z, 14), (z, 15)\}) \in F\,, \tag{9}$$

for some $t_4, t_5 \in \mathcal{T}$ and $k_4, k_5 \in \mathcal{K}$, where $t_4, t_5 > t_1, t_2, t_3$.

At the same time, core logic detects that all nodes have received packets 1..9, so a consistent global decision is made to deliver all of these, and communicated to lower layers; we model it as follows:

$$e_6 = (x, t_6, k_6, \{1..9\}) \in D\,, \tag{10}$$
$$e_7 = (y, t_7, k_7, \{1..9\}) \in D\,, \tag{11}$$
$$e_8 = (z, t_8, k_8, \{1..9\}) \in D\,, \tag{12}$$

for $t_6, t_7, t_8 \in \mathcal{T}$ and $k_6, k_7, k_8 \in \mathcal{K}$, where $t_6, t_7, t_8 > t_1, t_2, t_3$.

Once all forwarded packets arrive at z, this is reported as follows:

$$e_9 = (z, t_9, k_9, \{1..20\}) \in R\,, \tag{13}$$

for some $t_9 \in \mathcal{T}$ and $k_9 \in \mathcal{K}$, where $t_9 > t_4, t_5$ and $k_9 > k_3$.

Eventually, all nodes detect they have all packets up to 20, and this triggers three new events in D, permitting delivery on all nodes:

$$e_{10} = (x, t_{10}, k_{10}, \{1..20\}) \in D\,, \tag{14}$$
$$e_{11} = (y, t_{11}, k_{11}, \{1..20\}) \in D\,, \tag{15}$$
$$e_{12} = (z, t_{12}, k_{12}, \{1..20\}) \in D\,, \tag{16}$$

for $t_{10}, t_{11}, t_{12} \in \mathcal{T}$ and $k_{10}, k_{11}, k_{12} \in \mathcal{K}$, where $t_{10}, t_{11}, t_{12} \geq t_9$, $k_{10} > k_6$, $k_{11} > k_7$, and $k_{12} > k_8$. ■

Notice that our flows could generally be classified as inputs carrying information into the core logic layer or outputs carrying decisions out of it. The behavior of the core logic layer can be viewed as a distributed *function* continuously generating events in output flows from those in input flows (Figure 7). Each value at the output arises as a result of applying some operator to a set of values at the input. In the reliable multicast example just discussed, values in

D and F are computed from values in R. Each value is computed over past input: a value in D at time t can only be computed from values that f ow in R at times $t' < t$. Notice that this dependency is *distributed*, in the sense that the given value $D_x(i)$, generated on node x, does not depend only on values $R_x(j)$ f owing on the same node x, but also on values $R_y(k)$ that appear on other nodes $y \neq x$.

Like I/O automata (IOA) [20], 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 f ow and can be automatically translated into a scalable implementation. To this end, we introduce a set of functional building blocks: *aggregations*, *transformations*, *disseminations*, and *distributions*, and we gradually explain how to express multicast semantics as a composition of these (Figure 10).

2.2 Building Blocks

Aggregations. Flow β is an *aggregation* of f ow α if each value f owing in β can be expressed as a result of aggregating some set of values f owing in α, using some operator $\otimes$. Specif cally, for each event $e \in \beta$, there exists a f nite set of events $e'_1, \ldots e'_k \in \alpha$ such that the value $\nu(e)$ carried by e can be represented in terms of values $\nu(e'_i)$ carried by events e'_i in the following manner: $\nu(e) = \nu(e'_1) \otimes \nu(e'_2) \otimes \ldots \nu(e'_k)$. The process is illustrated on Figure 8.

For example, suppose that the token ring α on Figure 8 receives from its lower layers information about packets that are *stable* (i.e., received on all nodes) in some part of the network; this information is generated by lower layers on nodes a, b, and c. Ring α may circulate its token to collect the different values v_a, v_b, and v_c received on different nodes, calculate a set intersection $v_\alpha = v_a \cap v_b \cap v_c$, thus obtaining the set v_α of identif ers of packets stable in the entire region for which α is responsible, and pass v_α to upper layers, as a part of f ow α, in an event (x, t, k, v_α). In this case, f ow α is an aggregation over the union of all f ows entering ring α from below.

Formally, we require that for an associative commutative binary *operator* $\otimes : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$, the following condition holds:

$$\forall_{e \in \beta} \, \exists_{A \subseteq \alpha} \, (|A| < \infty) \wedge \left(\forall_{e' \in A} \, \tau(e') < \tau(e)\right) \wedge \ldots$$
$$\cdots \wedge \nu(e) = \bigotimes_{e' \in A} \nu(e') \,. \quad (17)$$

Operators that are associative and commutative and may be used in aggregations include, e.g., $\cup$, $\cap$, $+$, $*$, $\wedge$, $\vee$, min, and max (of course, we will use different operators for different types of values).

Note that as nodes join, leave or fail, and the hierarchy evolves, the set of nodes that contribute values to the aggregation can change. To more specif cally characterize the way aggregation proceeds, we def ne two families of partial functions: $\mu_x : \mathcal{K} \to \mathbb{P}(\mathcal{X})$, called *memberships*, and $\sigma^y_x : \mathcal{K} \to \mathcal{K}$, called *selectors*, for $x, y \in \mathcal{X}$. Formally, these two partial function families satisfy the following:

$$\text{dom}(\mu_x) = \{k \in \mathcal{K} \mid \beta_x(k) \text{ is def ned}\} \,, \quad (18)$$

$$\forall_{k \in \text{dom}(\mu_x)} \, |\mu_x(k)| < \infty \,, \quad (19)$$

$$\text{dom}(\sigma^y_x) = \{k \in \text{dom}(\mu_x) \mid y \in \mu_x(k)\} \,, \quad (20)$$

$$\beta_x(k) \text{ is def ned} \Rightarrow \beta_x(k) = \bigotimes_{y \in \mu_x(k)} \alpha_y(\sigma^y_x(k)) \,. \quad (21)$$

Their roles can be explained as follows. Choose any event $e \in \beta$. If β is an aggregation on α, there exist $e'_1, \ldots e'_n \in \alpha$ such that $\nu(e) = \nu(e'_1) \otimes \ldots \nu(e'_n)$. Each $e'_i \in \alpha$ f ows at some location $x_i = \chi(e'_i)$. We say that nodes $x_1, \ldots x_n$ have *contributed values* to e's aggregation. The membership functions μ_x determine what the locations are. Specif cally, if $\chi(e) = x$, and $\kappa(e) = k$, then the locations are $\{x_1, \ldots x_n\} = \mu_x(k)$ (Figure 9). Note that μ_x takes

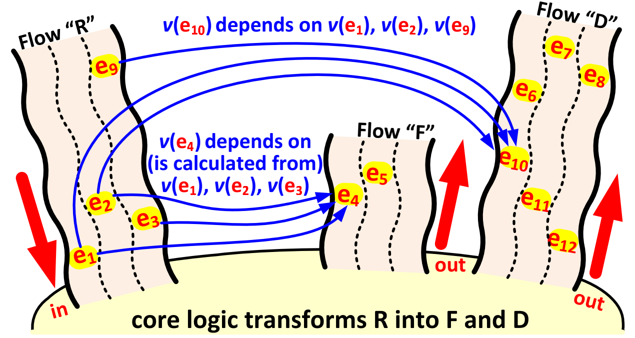

Figure 7: The core logic layer can be modeled as a distributed function that generates values in the output flows from values in the input flows. For example, the set of identifiers of packets ready for delivery $\nu(e_4)$ was calculated from the sets of identifiers of packets received: $\nu(e_4) = \nu(e_1) \cap \nu(e_2) \cap \nu(e_3)$.

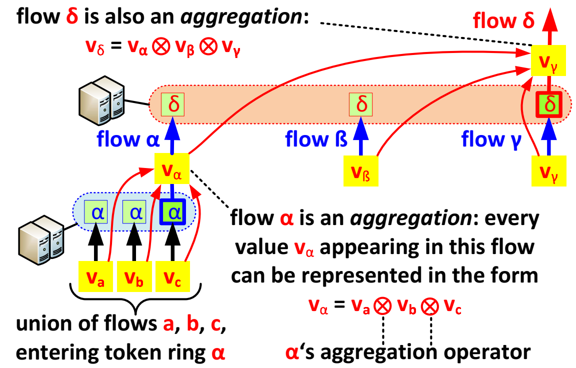

Figure 8: Flow α that token ring α sends to upper layers in the hierarchy is an aggregation over the flow that is obtained as a union of flows a, b, and c entering ring α from the lower layers. For each event $e_\alpha \in \alpha$, there are events $e_a \in a$, $e_b \in b$, and $e_c \in c$ such that $\nu(e_\alpha) = \nu(e_a) \otimes \nu(e_b) \otimes \nu(e_c)$.

version as argument. For any $e, e' \in \beta$ such that $\chi(e) = \chi(e')$ and $\kappa(e) = \kappa(e')$, the same nodes contribute values for e and e'.

The selector function is similar in spirit: now that we specif ed the nodes that contribute values, we go one step further and specify the versions of values they contributed. For each $y \in \mu_x(k)$, node y contributes a value with a local version $\sigma^y_x(k)$ to the aggregation.

Now, recall from Equation (3) that location and version together identify the value. Thus, membership and selector functions jointly determine all values in the aggregated f ow (through Equation (21)).

It should be noted that some aggregations (in the sense of Equation (17)) may be impossible to model via memberships μ_x and selectors σ^y_x. We refer to such aggregations as *irregular*. Our model is concerned mostly with *regular* aggregations that can be expressed via Equation (21). By placing various constraints on memberships and selectors, we can further distinguish different subclasses of regular aggregations, such as *coordinated* and *in-order* (Section 2.4). This allows us to formulate general theorems one can use to reason at a high, functional level about the behavior of concrete protocols.

Another point worth noting is that although Equation (21) refers to memberships $\mu_x(k)$, we only assume the *existence* of such sets. Unlike in most approaches, where nodes must *learn* global membership as part of the protocol, in our approach memberships $\mu_x(k)$ are never explicitly constructed and never materialize anywhere in the system. We discuss this aspect in more detail in Section 2.5.

In the example on Figure 10, there are three aggregations: S, H, and T. S is an aggregation on R because to find which packets are *stable* (flow S), the protocol intersects sets of identifiers of packets received by individual nodes (reported in R); this is symbolized by equation $S = \cap R$ in the wavy shape representing flow S (inequality $R_x \subseteq S$ following $S = \cap R$ is not a typo; it represents an extra *guarding condition* imposed on new nodes; this is discussed in Section 2.4). Likewise, $H = \cup R$ symbolizes the fact that to calculate the set of packets the protocol *heard of* (that were received by some nodes), we calculate the set union of values in R. Note that S and H are both aggregations over R, but they use different operators $\cap$ and $\cup$. Also, at this level we do not specify *where* flows S, H flow; presumably, aggregated values emerge at a designated leader – the root of the hierarchy. We discuss this in more detail in Section 2.5.

Flow M is more complex; it is similar to flow F in that events in M carry sets of forwarding requests of the form (x, i), but unlike F, which carries forwarding requests that will originate locally (the lower layers on the local node will forward data elsewhere), flow M carries requests that will terminate locally (other nodes will forward data to this node). Each node x generates events in M to express its own needs: the events carry sets of pairs (x, i), where i is the identifier of some packet that has not arrived at node x. Flow T, which represents a system-wide *todo* list (the set of all forwarding requests that need to be satisfied for all losses to be repaired) is an aggregation on M because to calculate the global pool of requests, we take the set union of requests generated by the individual nodes.

Transformations. Flow β is a *transformation* of flows $\alpha^1, \ldots \alpha^n$ if for each event $e \in \beta$, the value $\nu(e)$ can be expressed as a result of applying a certain n-argument function Ψ_x to a list of values appearing in flows α^i (one value from each). Note the similarity to aggregation: here again, we express values in flow β in terms of values in other flows; the difference is that, rather than taking multiple values from a single other flow α, we now take a single value from each of a list of flows (α^i), and instead of aggregating values with a binary operator, we feed them as arguments to function Ψ_x. Formally, for certain operators $\Psi_x : \mathcal{V}^n \rightarrow \mathcal{V}$ the following holds:

$$\forall_{e \in \beta}\, \exists_{e'_1 \in \alpha^1} \ldots \exists_{e'_n \in \alpha^n} \left(\forall_{1 \leq i \leq n}\, \tau(e'_i) < \tau(e)\right) \wedge \ldots$$
$$\cdots \wedge \nu(e) = \Psi_{\chi(e)}(e'_1, \ldots e'_n)\,. \quad (22)$$

As with aggregations, we distinguish a class of *regular* transformations that can be characterized through their membership functions $\mu^j_x : \mathcal{K} \rightarrow \mathcal{X}$ and selector functions $\sigma^j_x : \mathcal{K} \rightarrow \mathcal{K}$, for $x \in \mathcal{X}$ and $j \in \{1, 2, \ldots, n\}$, in the following manner:

$$\text{dom}(\mu^j_x) = \text{dom}(\sigma^j_x) = \{k \in \mathcal{K} \mid \beta_x(k) \text{ is defined}\}\,, \quad (23)$$
$$\beta_x(k) \text{ is defined} \Rightarrow \beta_x(k) = \Psi_x(v_1, v_2, \ldots, v_n) \ldots$$
$$\ldots \text{where} \quad \forall_{1 \leq j \leq n}\, v_j \stackrel{def}{=} \alpha^j_{\mu^j_x(k)}(\sigma^j_x(k))\,. \quad (24)$$

In the example on Figure 10, there is a single transformation M; it depends on flows R and H in the following manner. First, node x takes some value $R_x(k)$ from flow R; this is the local information about packets that were received. Node x also takes value $H_{x'}(k')$ from flow H; this is information about all packets that the protocol heard of (received by some nodes) that emerged somewhere in the system and was propagated to node x. After substracting, $H_{x'}(k') \setminus R_x(k)$ represents the set of identifiers of packets currently missing at x. After taking a cartesian product with its own identifier, $\{x\} \times (H_{x'}(k') \setminus R_x(k))$, node x produces a set of forwarding requests (x, i) for all packets i it has missed. This value is then carried as an event in flow M (aggregated into the global *todo* list in flow T). Formally, M can be described as a transformation on flows R and H using function $\Psi_x(v_1, v_2) = \{x\} \times (v_2 \setminus v_1)$. This is symbolized by equation $M_x = \{x\} \times (H \setminus R_x)$ at the center of Figure 10.

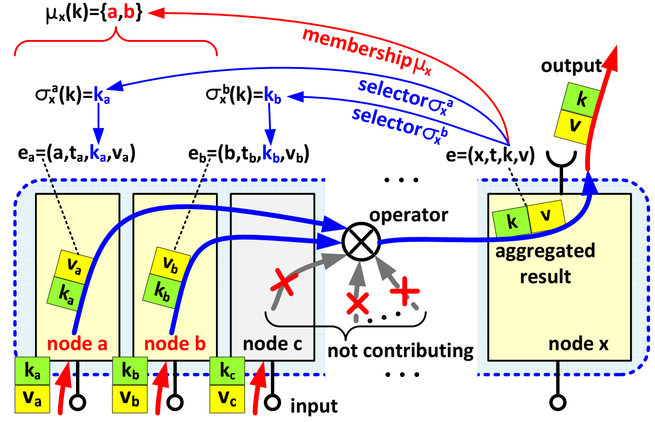

Figure 9: Event $e \in \beta$ occurs at location $x = \chi(e)$, with version $k = \kappa(e)$. Membership μ_x determines that its value, $v = \nu(e)$, must have been calculated from values that appear on nodes a and b (since $\mu_x(k) = \{a, b\}$). For each of those, selectors σ^a_x, σ^b_x determine the versions of the contributed events: $k_a = \kappa(e_a) = \sigma^a_x(k)$ and $k_b = \kappa(e_b) = \sigma^b_x(k)$. Thus, $\nu(e) = \alpha_a(k_a) \otimes \alpha_b(k_b)$.

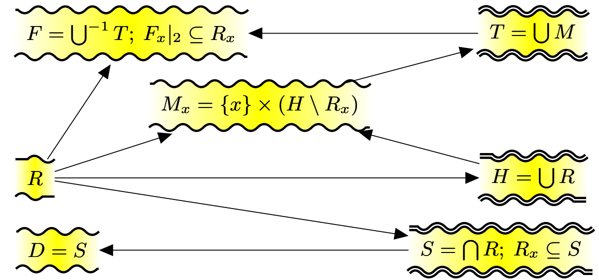

Figure 10: Multicast logic modeled as a graph of flow dependencies. Flows R, F, and D have been introduced earlier (Section 2.1). Flow S carries information about packets that are *stable* (received everywhere), flow H about packets that are *heard of* (received by some nodes), flow M about packets locally *missing* on some nodes, and T collects information about all missing packets into a global *todo* list of forwarding requests. Flows are represented as wavy shapes. Arrows and equations within the wavy shapes represent flow dependencies. How exactly each of the dependencies works is explained step by step in Section 2.2. Double boundary on T, H, and S indicates a system-wide state.

Disseminations. Flow β is a *dissemination* of α if each value appearing in β appeared previously somewhere in α; formally,

$$\forall_{e \in \beta}\, \exists_{e' \in \alpha}\, \tau(e') < \tau(e) \wedge \nu(e) = \nu(e')\,. \quad (25)$$

In the example on Figure 10, flow D is defined as a dissemination of flows S (the protocol delivers only packets known to be stable).

The subclass of *regular* disseminations can be characterized via memberships $\mu_x : \mathcal{K} \rightarrow \mathcal{X}$ and selectors $\sigma_x : \mathcal{K} \rightarrow \mathcal{K}$, as follows:

$$\text{dom}(\mu_x) = \text{dom}(\sigma_x) = \{k \in \mathcal{K} \mid \beta_x(k) \text{ is defined}\}\,, \quad (26)$$
$$\beta_x(k) \text{ is defined} \Rightarrow \beta_x(k) = \alpha_{\mu_x(k)}(\sigma_x(k))\,. \quad (27)$$

Distributions. The concept of *distribution* can be understood as the opposite of aggregation: flow β is a distribution over α if each value v in a subset $\alpha' \subseteq \alpha$ maps to a set V of values in β, $V \subseteq \mathcal{V}$, such that aggregating values from V using some operator $\otimes$ yields v. Formally, we assume the existence of a *distribution* function δ,

$\delta : \alpha' \to \mathbb{P}(\beta)$ for some $\alpha' \subseteq \alpha$, for which the following holds:

$$\forall_{e \in \alpha'} \ (|\delta(e)| < \infty) \wedge \nu(e) = \bigotimes_{e' \in \delta(e)} \nu(e') \ , \quad (28)$$

$$\forall_{e,e' \in \alpha'} \ (e \neq e' \Rightarrow \delta(e) \cap \delta(e') = \varnothing) \ , \quad (29)$$

$$\beta = \bigcup_{e \in \alpha'} \delta(e) \ . \quad (30)$$

One can define the aggregation that given distribution is the inverse of by expressing μ and σ in terms of δ (details omitted for brevity).

In the example on Figure 10, flow F is a distribution of T. Recall that values $T_x(k)$ represent global sets of forwarding requests (system-wide *todo* lists). Whenever a to-do list $T_x(k)$ flows in T, it is partitioned (distributed) among the available nodes, so that they can be satisfied in parallel. On different nodes $x_1, \ldots x_n$, values $F_{x_1}(k_1), \ldots F_{x_n}(k_n)$ are sent to the lower layers to request forwarding so that $F_{x_1}(k_1) \cup \cdots \cup F_{x_n}(k_n) = T_x(k)$. Flow T serves as a global "hub" through which forwarding requests issued by nodes missing packets (flow M) are routed to nodes that satisfy them (flow F). (The protocol does not need to *physically* tunnel all forwarding requests through a single root; this is just a functional, *logical* view.) Condition $F_x|_2 \subseteq R_x$ on Figure 10 symbolizes extra requirement: the requests are routed to nodes that can satisfy them.

2.3 Strong Semantics

In our model, stronger semantics can be expressed by defining a predicate $\phi : \mathcal{V} \to \mathbb{B}$ (where $\mathbb{B} = \{false, true\}$), and requiring that if $\phi(v)$ holds for some value v anywhere in the given flow, then at each non-faulty location eventually $\phi(v')$ holds for some $v' \in \mathcal{V}$.

Formally, flow α is *atomic* with respect to predicate ϕ and a set of locations $S \subseteq \mathcal{X}$ (or ϕ-*atomic* on S) if the following holds:

$$(\exists_{e \in \alpha} \ \phi(\nu(e))) \Rightarrow \forall_{x \in S \cap \mathcal{X}_0} \exists_{e' \in \alpha} \ \chi(e') = x \wedge \phi(\nu(e')) \ . \quad (31)$$

One can capture many types of strong semantics as a requirement that certain flows are ϕ-atomic, for some predicate ϕ. Intuitively, ϕ represents a progress condition, and we require that once progress is achieved anywhere in the system, it is not lost, and it is eventually consistently reproduced at all non-faulty nodes from a certain set S.

In the example on Figure 10, we want to ensure that if any node (even a faulty one) delivers packet i, eventually all non-faulty nodes deliver it. We can express this using Equation (31) by substituting $S \equiv \mathcal{X}, \alpha \equiv D$, and $\phi(v) \equiv$ (if $i \in v$ then $true$, otherwise $false$).

In asynchronous systems, of course, such properties are impossible to guarantee unconditionally [7]; typically, they are conditional on the existence of an appropriate failure detector [3]. 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 reliably affects all future decisions made by the protocol. The key to understanding our approach lies in the different way information is *remembered* in our system. Instead of relying on global views and state transfer, we require that certain flows be *monotonic*. Monotonicity alone does not yet imply Equation (31), but it does so if combined with liveness properties.

Monotonicity. Flow α is called (*strongly*) *monotonic* if events with higher versions also have larger values (with respect to $\leq_{\mathcal{V}}$). Formally, for all pairs of events $e, e' \in \alpha$, the following must hold:

$$\kappa(e) \leq \kappa(e') \Rightarrow \nu(e) \leq \nu(e') \ . \quad (32)$$

Flow α is *weakly monotonic* if the following holds for all $e, e' \in \alpha$:

$$\chi(e) = \chi(e') \wedge \kappa(e) \leq \kappa(e') \Rightarrow \nu(e) \leq \nu(e') \ . \quad (33)$$

It is easy to see that a strongly monotonic flow is always consistent.

In our example protocol on Figure 10, we need flows S and D to be strongly monotonic because each decision to deliver packets has permanent consequences not only to the node that delivered the packet, but to the entire group: it forces other nodes to do the same. The fact that S is monotonic places certain constraints on the way aggregation S is performed: a newly joining node that misses most packets cannot immediately participate in aggregation, for it would violate the monotonicity guarantee (we discuss this in Section 2.4).

Before explaining how to implement monotonicity, let's analyze its role in achieving stronger semantics. We need a few definitions.

Flow α is said to be *fresh* on a set of locations $S \subseteq \mathcal{X}$ if whenever an event with a new version appears anywhere in α, eventually an event with the same or newer version appears at every non-faulty location in S. Formally, the following must hold:

$$\forall_{e \in \alpha} \forall_{x \in S \cap \mathcal{X}_0} \exists_{e' \in \alpha} \ \chi(e') = x \wedge \kappa(e) \leq \kappa(e') \ . \quad (34)$$

Predicate $\phi : \mathcal{V} \to \mathbb{B}$ is said to be *monotonic* if the following holds:

$$\forall_{v,v' \in \mathcal{V}} \ \left(\phi(v) \wedge v \leq v' \Rightarrow \phi(v')\right) \ . \quad (35)$$

One example of a monotonic predicate is the delivery condition for D discussed earlier, $\phi(v) \equiv$ (if $i \in v$ then $true$, otherwise $false$).

THEOREM 2.1. *If flow α is fresh and monotonic, then it is also ϕ-atomic for every monotonic predicate $\phi : \mathcal{V} \to \mathbb{B}$.*

The proofs of this and all other theorems formulated in this paper can be found in the Appendix.

Intuitively, *freshness* expresses a certain type of a liveness condition: if any part of the system makes progress, all non-faulty nodes must eventually also make equivalent amount of progress; however, it does not determine the exact nature of that progress. Conversely, *monotonicity* constraints the types of progress that can be made. In practice, freshness and monotonicity are achieved through different (and to some degree complementary) mechanisms.

In the remainder of this paper, we focus on achieving monotonicity. Specifying and proving liveness is known to be a hard problem [4], and it requires more formal apparatus than what we have introduced in this paper. We will discuss liveness in our future work.

The reader may have noticed that failure handling is not explicit in our work; indeed, it is implicit in the definition of monotonicity. If node x fails right after value $\beta_x(k)$ flows at it, monotonicity still constraints values $\beta_y(j)$ at nodes $y \neq x$. The protocols that implement 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 group individually handles its own failures; each flow is thus *individually* subject to the FLP result [7]. In reality, no flow can (unconditionally) be monotonic and fresh at the same time.

2.4 Achieving Monotonicity

The most important result of this section is stated in Theorem 2.2, which shows how to implement strong monotonicity using simpler properties, and that motivates the aggregation protocol presented in Section 3.2. Before stating the theorem, we need a few definitions.

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

$$v_1 \leq v_2 \Rightarrow v_1 \otimes v_3 \leq v_2 \otimes v_3 \ , \quad (36)$$

$$v_1 \otimes v_2 \leq v_1 \ . \quad (37)$$

Many operators are lower bounds, but with respect to different orders. For example, $\cap$ is a lower bound if $v \leq v' \equiv v \subseteq v'$, whereas for $\cup$, the opposite order must be employed, i.e., $v \leq v' \equiv v' \subseteq v$.

Aggregation β is *coordinated* if memberships μ_x and selectors $\sigma_{x'}^y$ are identical for different x, x', i.e., if the following holds:

$$\forall_{x,x' \in \mathcal{X}} \forall_{k \in \text{dom}(\mu_x) \cap \text{dom}(\mu_{x'})} \; \mu_x(k) = \mu_{x'}(k) \; , \quad (38)$$

$$\forall_{x,x',y \in \mathcal{X}} \forall_{k \in \text{dom}(\sigma_x^y) \cap \text{dom}(\sigma_{x'}^y)} \; \sigma_x^y(k) = \sigma_{x'}^y(k) \; . \quad (39)$$

It is not hard to see that every coordinated aggregation is consistent.

Aggregation β is *in-order* if its selectors σ_x^y satisfy the following:

$$\forall_{x,x',y \in \mathcal{X};\; k \in \text{dom}(\sigma_x^y);\; k' \in \text{dom}(\sigma_{x'}^y);\; k \leq k'} \;\; \sigma_x^y(k) \leq \sigma_{x'}^y(k') \; . \quad (40)$$

Both properties are easy to satisfy by aggregating in rounds, and so that each node always contributes the latest value it received in α.

Aggregation β on α is called *guarded* if two conditions are satisfied: (i) the set of versions that appear in β is isomorphic with the set of natural numbers $\mathbb{N}$ (so for each version there are finitely many versions smaller than it), and (ii) for each pair of aggregated values with subsequent versions $k < k'$, and for each new node y participating only in the more recent aggregation, the value $\alpha_y(\sigma_{x'}^y(k'))$ contributed by y is not smaller than either the full or partial result of the former aggregation. Let $\mathcal{K}(\beta)$ be the set of versions that appear in β, $\mathcal{K}(\beta) = \{\kappa(e) \mid e \in \beta\}$. Formally, we require the following:

$$\forall_{k \in \mathcal{K}(\beta)} \left| \{k' \in \mathcal{K}(\beta) \mid k' < k\} \right| < \infty \; , \quad (41)$$

$$\begin{aligned} &\forall_{x,x',y \in \mathcal{X}} \forall_{k,k' \in \mathcal{K}} \; \big(\beta_x(k) \text{ and } \beta_{x'}(k') \text{ are defined}\,\big) \wedge \ldots \\ &\cdots \wedge \big(k < k' \wedge \neg \exists_{k'' \in \mathcal{K}(\beta)} \, k < k'' < k'\big) \wedge \ldots \\ &\cdots \wedge \big(y \in \mu_{x'}(k') \setminus \mu_x(k)\big) \Rightarrow \ldots \\ &\cdots \Rightarrow \exists_{N_p \subseteq \mu_x(k)} \; \big(\alpha_y(\sigma_{x'}^y(k')) \geq \bigotimes_{z \in N_p} \alpha_z(\sigma_x^z(k))\big) \; . \quad (42) \end{aligned}$$

This means that every node joining the aggregation must obtain at least a partial result of the current, or of the immediately preceding aggregation before its own value can be included. The aggregation protocol in Section 3.2 was designed to satisfy this requirement. In general, this property forces nodes joining the group that performs aggregation to wait for 1-2 protocol rounds until they can synchronize with the rest of the group. In a sense, this is a bit similar to state transfer. Membership services need not participate in the process.

Dissemination β is *in-order* if selectors σ_x satisfy the following:

$$\forall_{x,x' \in \mathcal{X};\; k \in \text{dom}(\sigma_x);\; k' \in \text{dom}(\sigma_{x'});\; k \leq k'} \;\; \sigma_x(k) \leq \sigma_{x'}(k') \; . \quad (43)$$

THEOREM 2.2. *If flow β is a guarded, coordinated and in-order aggregation over flow α, using a lower-bound, idempotent operator $\otimes$, and α is weakly monotonic, then β is strongly monotonic.*

THEOREM 2.3. *If flow β is an in-order dissemination of a monotonic flow α, then β is also monotonic.*

In the example on Figure 10, R is weakly monotonic, and operator $\cap$ is an idempotent lower bound, hence aggregation S is monotonic *if only* it is coordinated, in-order, and guarded, and dissemination D is monotonic *if only* it is in-order.

2.5 Hierarchical Composition

This section presents the central result of this paper: Theorem 2.4 that underpins the architecture from Figure 2. It proves that monotonicity, which, as explained in Section 2.3, can be used to achieve strong semantics, is possible to construct in a hierarchical manner, layer by layer, and under fairly weak conditions; in particular, this theorem implies that coordination between local MSs is not needed.

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

We assume 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, where non-minimal β can have infinitely many children.

THEOREM 2.4. *Every sink in an aggregation network H is an aggregation on the union of all sources $\bigcup \bot^H$. If all sources are weakly monotonic and all non-minimal $\beta \in H$ are guarded coordinated in-order aggregations on their respective $\bigcup \mathcal{C}(\beta)$ with idempotent lower-bound $\otimes$, then all sinks $\beta \in \top^H$ are monotonic.*

Now, we will explain the practical significance of this theorem by referring to the architecture on Figure 2 and our reliable multicast example. Suppose that the leaf-level components (a, b, and so on) on Figure 2 produce a weakly monotonic flow, such as R, and that aggregations in all token rings satisfy the requirements of this theorem; e.g., they are all coordinated, in-order, guarded, using operator $\cap$. Theorem 2.4 ensures that the aggregate flow $S = \cap R$ that emerges at the root of the hierarchy is strongly monotonic if only token rings are well-ordered in terms of their parent-child relationships. Rings might fail and reconfigure, and nodes might join and fail, and the entire hierarchy might evolve dynamically, but as long as components joining each token ring in the hierarchy on Figure 2 wait to learn the results of prior aggregations, monotonicity is not violated. Now, we only need a mechanism that ensures the well-ordering of token rings. The core ideas are outlined in Section 3.1.

3. ARCHITECTURE

3.1 Hierarchical Delegation

We begin by discussing the internal structure of the client's protocol stack. The stack includes the three components shown on Figure 5, including the *working component* representing lower layers, and the *data flow component* P implementing the core logic. Interaction between the working component and P is done via events that contain values tagged with version numbers (Figure 11).

Initially, P contains no actual protocol logic, and does not know what to do with the values (such as $R_x(k)$) it receives; it only contains a *bootstrap code* for contacting a *delegation authority* (DA). DA is a small service that upon request returns a serialized description of P's protocol stack. It hands out the same code to all clients.

There are two classes of DAs: the *root authority* (RA) and all the rest. The RA returns a *root code* that does not involve any interaction with other nodes; it simply consumes values, performs internal computations (transformations on flows), 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, when it may be migrated).

A regular, non-root DA returns an *aggregation* code that implements a token ring protocol running among all clients boostrapped from this DA (Figure 12). Dynamic protocol stack compositions are facilitated by our *Live Distributed Objects* platform [1, 27], and the token ring protocols are similar to our prior work on QSM [26].

The aggregation component (described in Section 3.2) uses this token ring to aggregate and disseminate values in the local group; it corresponds to a single ring in the hierarchy on Figure 2. The group uses a private membership service (MS) to self-organize. Each DA manages only a small subset of clients, so the local MSs should not experience heavy load. The code for contacting local MS, address, parameters and so on, are all embedded in code returned by the DA.

Delegation authorities form a hierarchy: each DA except for RA has a *parent*, DA'. P's aggregation code includes a recursively embedded data flow component P', which is configured to bootstrap from DA' (Figure 12). Normally, P' stays dormant. It can prefetch its own code from DA', but does not activate the downloaded code, and P does not attempt to interact with it. P' remains 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. Thus, token ring leaders form higher-level token rings, exactly as postulated in the architecture diagram on Figure 1.

The above pattern now repeats recursively: P_1 contains an embedded P_2, which contains P_3, and so on (Figure 13). If the local node happens to be a leader in each ring it is a part of, this recursion terminates with the inner-most component boostrapped from RA; the node then becomes the root of the hierarchy, and makes global decisions for the entire system. Otherwise, the inner-most P_k is running aggregation code while its embedded P_{k+1} stays dormant. The node then serves as the root of some subtree in our hierarchy.

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 cross-layer interaction is, when a data flow component P_k on a client activates, disposes, or exchanges values with the component P_{k+1} embedded in it. Different MSs and DAs need not interact with one-another.

The hierarchy of DAs emerges via the following process. First, the RA is created, and configured to return the root code. A single top-level DA is also created with its associated MS; aggregation code P it returns is configured to bootstrap embedded P_{k+1} from RA. All nodes bootstrapped from the top-level DA become members of the top-level ring and one of them runs the root code. This lays the foundation. The process 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 11), the list of all aggregation rules (specifications for the component named "aggregation" on Figure 12), and a bootstrap code for the DA that issued the invitation. Invitations can be disseminated, e.g., via email.

An invitation can be "consumed" directly by a client, by assembling its parts into a protocol stack (Figure 11, Figure 12). Alternatively, the invitation can be used to setup a new DA, which assumes the DA that issued the original invitation to be its own parent. The new DA can now issue its own invitations, and make children, by replacing the bootstrap code in the parent's 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 2.4, it suffices that DAs form a tree and never change their parents.

The correctness of behavior under churn follows from our theorems. When a node x fails, the MSs of all rings it was a member of disseminate new membership views (without x) to repair the rings. Each of the affected rings chooses a new leader (if necessary) and circulates 1-2 tokens to ensure that information about the preceding rounds is properly detected and replicated; it then simply resumes its work. New leaders may need to activate their embedded aggregation components and join higher-level rings; if so, they undergo a join protocol discussed in Section 3.2. There is no need for cross-

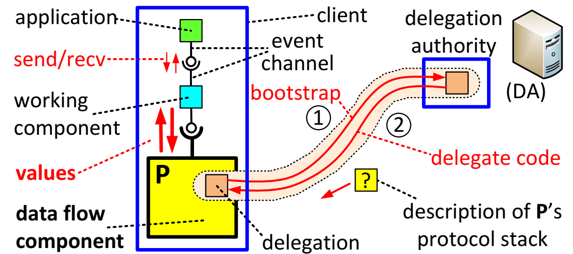

Figure 11: The structure of the client's protocol stack: P's code is bootstrapped from the DA (as soon as P becomes activated).

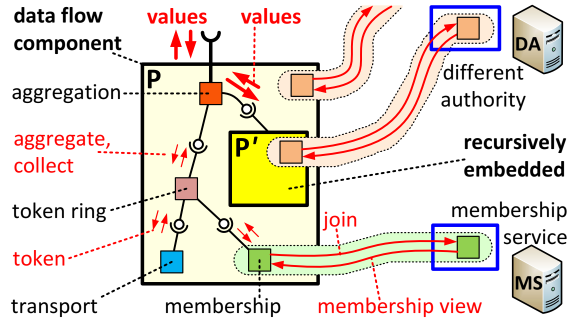

Figure 12: The internal structure of the data flow component P running aggregation code (bootstrapped from a non-root DA).

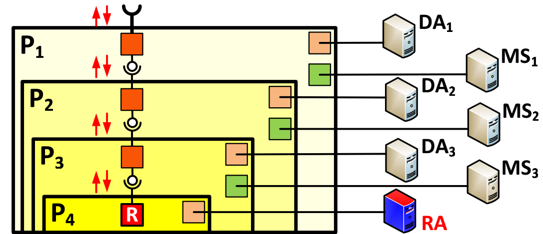

Figure 13: A recursively built stack of aggregation components.

level coordination in this architecture. Each token ring only needs to ensure that its own aggregations are coordinated, in-order, and guarded, its own failures repaired, and that its own local members subscribe to the higher-level rings when they become ring leaders.

3.2 Aggregation Component

Aggregation components (Figure 13) interact using *value buckets*; one bucket for each input or output flow (Figure 14). When a value arrives from a component higher or lower in the hierarchy, it goes into an input bucket, and when a value in some output bucket changes, it is sent out. Internally, each value change triggers *aggregation rules* that update other buckets. All components (the reader may think of them as the green boxes in the hierarchy on Figure 2) except the root run the *regular* rules, the lowest-level components additionally run the *client* rules, while the root runs only *root* rules.

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

Values are aggregated by passing tokens around the ring. A 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 deposited in the output buckets of other nodes. Thus, round after round, the ring collects values from input buckets of all ring members, aggregates them, tags the result with a new version, and replicates it all over the ring. This is the normal case.

Nodes that wish to join the ring (e.g., because they are just entering the system, or because they have just become leaders in lower-level rings and are required to join higher-level token rings by the rules discussed in Section 3.1) behave a little differently. Initially, they do not participate in aggregation, i.e., they passively observe tokens passing by, but do not change their contents (except for aggregations that are not guarded); this lets them gradually catch up with the rest: obtain state transfer, participate in loss recovery, etc. This is a *candidate* status. To become a *regular* member that fully participates in the protocol, 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 carrying a partial result v of the current, and some (even a partial) result v' of the preceding aggregation passes through a candidate, the candidate tests if $v' \leq v''$ holds, where v'' is the candidate's own value (the value it would contribute). If it does, the candidate can replace v with $(v \otimes v'')$, but it does not yet become a regular member. Instead, it records version k of the current aggregation, and waits for the following round. Only after a new token arrives with the result of k-th aggregation, the candidate promotes itself to the regular status. If it later f nds that it has been dropped from the view, it goes back 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 new node waits until it learns results of the immediately preceding round and ensures that the guarding condition holds. Once the node f nds out that its local value affected the result, this is no longer needed. Aggregation is coordinated and in-order: it is done in rounds and values put into buckets are ones with the highest versions ever seen. Thus, our protocol satisf es all of the requirements for Theorem 2.2.

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 signif cance of these factors in pure form, undisturbed by performance of other mechanisms, such as packet forwarding or state transfer, we use simplif ed 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 have been simulated: UDP transmissions, failures, and reboots, instead of using socket API, schedule events in the simulator, whereas all other components run normally. Clients still communicate via asynchronous events, establish connections, form rings based on membership updates, and serialize transmitted packets. Average network latency is set fo 10ms, and the rings circulate 10 tokens/s on average.

4.1 Aggregation in the Presence of Churn

In the f rst experiment, we cause clients to synchronously enter subsequent phases of processing. The integer-valued input f ow L informs the protocol of the latest phases $L_x(k)$ entered by each of the clients, and the output f ow N instructs each client which phase $N_x(k)$ to execute next. This mini-protocol can be concisely written as $L' = \min L;\ N = L' + 1$. Monotonic aggregation L' computes

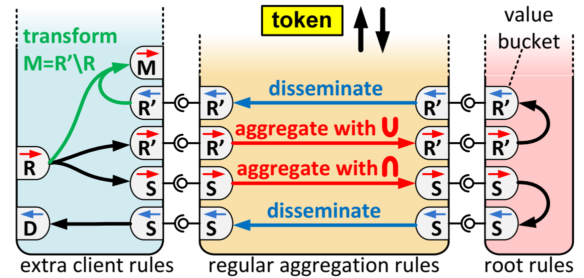

Figure 14: Example aggregation rules (a subset of rules shown).

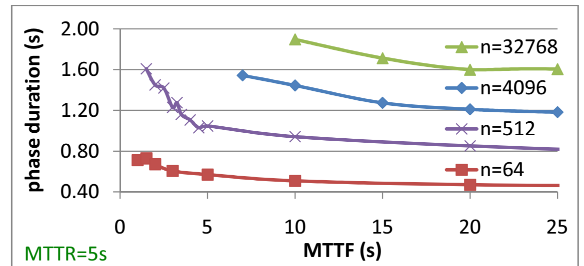

Figure 15: Phase duration as a function of system size and mean time to failure (MTTF).

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 can 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. MTTR/MTTF are chosen to be very small, at the order of 10s of seconds; this puts extra stress on our protocol. In reality, we expect churn to be much lower; this experiment is designed to push our architecture to the limit. Rebooted clients join under new identity, and undergo the guarded aggregation protocol of Section 3.2. The token ring size is 8 nodes on average.

The results on Figure 15 suggest that latency increases as a logarithm of system size (n); this is exactly what we would expect in a balanced hierarchy given that each ring works independently. It takes about 4 additional token rounds for each layer in the hierarchy (2 rounds each way), across a wide range of churn rates. Even under extreme churn (MTTF=10s), latency grows only by 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 amortized across the system. Notice 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 reconf guration. In our system, reconf guration 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 benef ts of hierarchically decomposing the GMS into multiple MSs are thus evident.

Naturally, this result depends on the fact that communication in different parts of the network can happen in parallel, and the aggregate system bandwidth is not limited; this may not be true in a data center. However, the same issue occurs with all peer-to-peer protocols run in such setting. In our system, the aggregate bandwidth of the control traff c is $\mathcal{O}(1)$ packets per node per second (KR with token rate R and the average ring size K), asymptotically optimal.

4.2 The Overhead of Value Representation

In the preceding experiment, all values carried by tokens would f t in a constant amount of space. In many real protocols, this is not the case (for example, when values are sets of packet identif ers). To bound resource usage, we have to limit token sizes, and truncate values that cannot f t. As a result, smaller batches of events can be handled in parallel, and consequently, the system slows down.

To illustrate this, in the second experiment we run a simplif ed commit protocol: each client receives transactions at a f xed rate, and independently decides to commit or abort, with probability adjusted so that a fraction p of transactions commit globally. Values in input f ows C, A are sets of identif ers of transactions that the individual nodes wish to commit ($C_x(k)$) or abort ($A_x(k)$). Output f ows C', A' carry global decisions. An internal f ow D records identif ers of transactions for which decisions have been made. The protocol can be written as $C' = \bigcap(C \setminus D)$; $A' = \bigcup(A \setminus D)$; $D = C' \cup A'$. Aggregations C' and A' are guarded and in-order.

Each value, as a set of numeric transaction identif ers, is encoded as a tuple of the form $((a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k), c)$, in which 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 undef ned. Operators $\cup$ and $\cap$ are modif ed 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) do not f t within the limit k_{max} and c may become lower. Because of this, a single aggregation may no longer suff ce to propagate all information from clients to the root.

In the f rst scenario in this experiment, we f x 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 16). 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 it is the serialization cost. As tokens grow, we need more CPU, but not extra rounds. Only if event rate exceeds1050 TPS (transactions per second), k_{max} is reached, values are truncated aggressively, transactions pile up, and latency shoots to inf nity (not shown).

In the second scenario, the rate is f xed at 1000 TPS, $p = 95\%$, and we vary system size (Figure 17). Latency and token size grow only logarithmically, and overall latency is nearly the same as in the previous experiment. Thus again, as long as the size of an average value that is being aggregated remains beneath the k_{max} threshold, the system responds to the increased load by increasing the token sizes, and latency remains essentially unaffected. At $\approx$32K nodes we are 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 f nd how much data would otherwise be truncated (Figure 18). We f nd that if 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 traff c in every ring, so 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.

4.3 Hierarchy Depth and Aggregation Rate

To conclude, we study the effect of varying token rate (Figure 19) and ring size (Figure 20). Having several tokens chase each other (e.g. >12 tokens/s in an 8-node ring with $\approx$10ms latency) results in

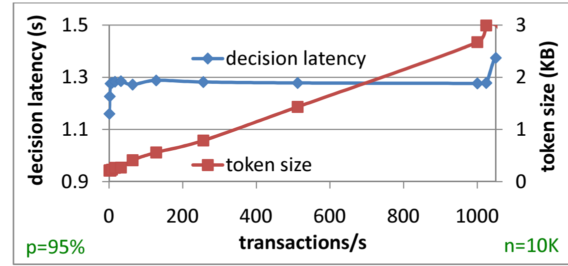

Figure 16: Decision latency and token size as functions of the frequency of application events (*transactions per second*, TPS).

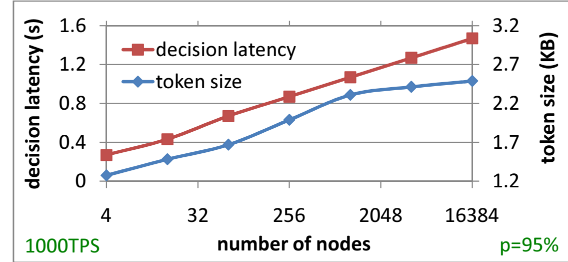

Figure 17: Latency and token size as functions of system size.

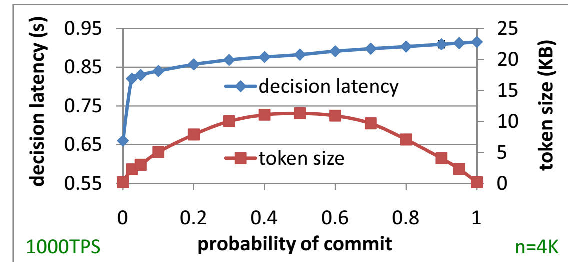

Figure 18: Latency and space overhead when aggregated data is not truncated ($k_{max} = \infty$), with 4096 clients and 1000 TPS.

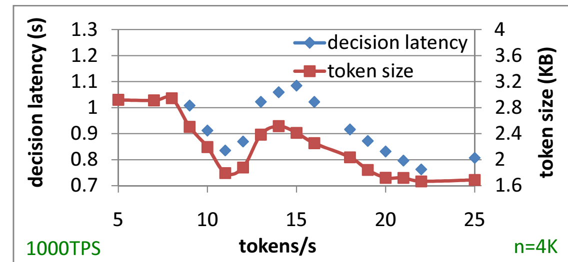

Figure 19: Varying the rate at which the tokens are circulating.

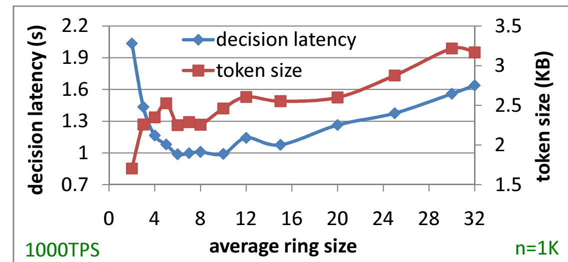

Figure 20: Varying the average size of rings (hierarchy fanout).

redundant work. Wrong ring size hurts latency, for either the hierarchy is deep, or it takes long to aggregate in each ring. A bad choice of parameters can affect latency by a factor of 2. Replacing rings with trees may help, but in practice, ensuring that the hierarchy is balanced appears to be the key challenge. Although our architecture is f exible, it does not allow protocols to change parents in the hierarchy, making algorithms for self-balancing trees harder to use.

5. RELATED WORK

Whereas many protocols are described in pseudocode and proven using temporal logic, I/O automata (IOA) [20] pioneered an approach that treats distributed protocols as components that operate on event streams. The components are modeled as *finite state machines* (FSMs): f nite automata, with transitions triggered by timeouts, message receipts, or application requests. IOA has been used to explore a number of protocols with strong reliability properties, for example in the work on Ensemble [12]. TLA [18] is another major model in this space with similar expressiveness; it has been used to model distributed consensus. FSM also served as a foundation for many protocol languages that can be compiled into executable code, e.g., MACE [17], and for recent SOA/WS-* standards for modeling peer-to-peer interactions, particularly WSCL [2].

The key difference with respect to our model is that whereas the above approaches focused on the compositional structure of protocols within individual endpoints, our model is more data-centric. We focus on composition of entire f ows. Eliminating node-centric aspects creates f exibility and freedoms, e.g., to build a hierarchy independently from the method of aggregation, and to batch events and exchange information in ways convenient to the runtime system. One manifestation of the power of our approach is evident in Theorem 2.4. Implementations based on our approach can also use these freedoms to achieve scalability, and to adapt to their environments, e.g., by switching between ring- and tree-based aggregation.

Among the existing non-FSM formalisms and languages, such as those based on CSP [13], most are too weak to express semantics such as atomicity and agreement [9]. The same is true of the more recent work on declarative networking (P2) [19], based on a version of the Datalog language; it can be hard to capture strong semantics without concepts such as consistent aggregation and membership built into the model. Indeed, P2 has been used mostly for loosely-coupled systems, such as peer-to-peer overlays, DTHs, and routing.

Data f ows in the sense of asynchronous, highly parallel pipelined processing have a long tradition in areas such as VLSI and DBMS. In those settings, membership is typically f xed in advance, but data f ow pipeline techniques have also been applied to networking, e.g., in Click [23], and distributed computing, e.g., in P2. Flows in those systems, however, are not *distributed* in the same sense as def ned in Section 2.1; they are point-to-point event streams, and transformations on them are local. Although distributed query engines such as Gamma [6], Volcano [10], or PIER [14] support hierarchical aggregation, they were designed for data mining, not for coordination, and lack strong properties similar to those discussed in Section 2.3.

There has been much research on hierarchical in-network aggregation in sensor networks that used trees and DAGs with redundant paths [21], gossip [11], or both [22]. Most work focuses on simple aggregates such as SUM, but some [30] supports medians, majority values, etc. Reliability in this context usually means accounting for nearly all sensor readings and ignoring duplicates in the presence of faults, and in some cases preserving integrity of the result in the presence of malicious nodes [8]. Besides redundancy, researchers have explored hashing [5] and model-based error correction [24]. Much work focuses on clustering schemes for building hierarchies in a way that trades performance for energy-eff ciency, e.g., [15].

Unlike the work on sensor networks, our model is not concerned with energy eff ciency; we assume full connectivity, and the ability to rely on external services. At the same time, our work targets considerably stronger semantics that those offered by typical sensor network aggregation schemes. In the models of interest to us, it does not suff ce for aggregated values to be approximately or probabilistically accurate; the desired semantics are def ned in terms of dynamic membership with crash failures. Nevertheless, some of these clustering techniques could be employed in our system, in combination with the delegation architecture described in Section 3.1. Our guarded aggregation and join protocol could also be viewed as having connections to model-based error correction [24].

Much research focused on making group membership protocols scalable, in particular also through the use of hierarchy [16], but scalability in traditional GMS-driven protocols is ultimately limited by the inherently non-scalable requirement that all group members receive the full global view. Our approach relaxes this requirement.

6. CONCLUSIONS

We proposed a novel approach to building protocols with strong properties that does not rely on global membership. We developed a theory to reason about our model, a supporting architecture, and we brief y reported on the performance of our initial prototype. Our approach appears to be fairly general, scalable, and churn-tolerant.

7. REFERENCES

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

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

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *JACM*, 1996.

[4] G. Chockler, I. Keidar, and W. Vitenberg. Group communication specif cations: A comprehensive study. *ACM Computer Surveys, 33(4):1, pp. 43, Dec 2001.*, 2001.

[5] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. *ICDE 2004*.

[6] D. DeWitt et al. Gamma - a high performance dataf ow database machine. *VLDB*, 1986.

[7] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 1985.

[8] K. Frikken and J. Dougherty. An eff cient integrity preserving scheme for hierarchical sensor aggregation.

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

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

[11] I. Gupta, R. v. Renesse, and K. Birman. Scalable fault-tolerant aggregation in large process groups. *DSN 2001*.

[12] J. Hickey, N. Lynch, and R. van Renesse. Specif cations and proofs for ensemble layers. *TACAS*, 1999.

[13] C. Hoare. Communicating sequential processes. *CACM*, 21(8), 1978.

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

[15] W.-S. Jung, K.-W. Lim, Y.-B. Ko, and S.-J. Park. A hybrid approach for clustering-based data aggregation in wireless sensor networks. *ICDS 2009*.

[16] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for wans. *TOCS*, 2002.

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

PLDI, 2007.
[18] L. Lamport. The temporal logic of actions. *TOPLAS*, 1994.
[19] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SOSP*, 2005.
[20] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *PODC*, 1987.
[21] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SOSP 2002*.
[22] A. Manjhi, S. Nath, and P. Gibbons. Tributaries and deltas: eff cient and robust aggregation in sensor network streams. *SIGMOD 2005*.
[23] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SOSP*, 1999.
[24] S. Mukhopadhyay, D. Panigrahi, and S. Dey. Model based error correction for wireless sensor networks. *SECON 2004*.
[25] K. Ostrowski, K. Birman, and D. Dolev. Programming Live Distributed Objects with Distributed Data Flows. Cornell University Tech Report. http://hdl.handle.net/1813/12766.
[26] K. Ostrowski, K. Birman, and D. Dolev. QuickSilver Scalable Multicast (QSM). *NCA*, 2008.
[27] K. Ostrowski, K. Birman, D. Dolev, and J. Ahnn. Programming with Live Distributed Objects. *ECOOP*, 2008.
[28] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *JSAC*, 1997.
[29] F. Schneider. Byzantine generals in action: implementing fail-stop processors. *TOCS*, 1984.
[30] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. *SenSys 2004*.

APPENDIX

A. PROOF OF THEOREM 2.1

PROOF. Suppose $\phi(\nu(e)) = true$ for some $e \in \alpha$, and let $x \in S \cap \mathcal{X}_0$. Since α is fresh, $\exists_{e' \in \alpha}\, \chi(e') = x \wedge \kappa(e) \leq \kappa(e')$. Since α is monotonic, $\kappa(e) \leq \kappa(e')$ yields $\nu(e) \leq \nu(e')$, and since ϕ is monotonic, $\phi(\nu(e)) \wedge \nu(e) \leq \nu(e')$ yields $\phi(\nu(e'))$. □

B. PROOF OF THEOREM 2.2

PROOF. Let $\beta_x(k), \beta_{x'}(k')$ be any values that f ow in β such that $k \leq k'$. We need to show that $\beta_x(k) \leq \beta_{x'}(k')$. If $k = k'$, this follows trivially from the fact that β is coordinated. Suppose $k < k'$. If k is the last version preceding k' in β, we follow the reasoning below. If not, then from the fact β is guarded, there exists a f nite chain of versions $k = k_1 < k_2 < \cdots < k_n = k'$ in $\mathcal{K}(\beta)$ such that k_i is the last version preceding k_{i+1} in β for all $1 \leq i < n$ and a corresponding set of locations such that $\beta_{x_i}(k_i)$ is def ned for $i < n$; we then follow the reasoning below for each $i < n$ to show $\beta_{x_i}(k_i) \leq \beta_{x_{i+1}}(k_{i+1})$, and get $\beta_x(k) \leq \beta_{x'}(k')$ by transitivity.

So, let's suppose k directly precedes k' in $\mathcal{K}(\beta)$. Let's partition nodes involved in aggregations into three groups: let $N_o = \mu_x(k) \setminus \mu_{x'}(k')$ be nodes involved in just the older one, $N_n = \mu_{x'}(k') \setminus \mu_x(k)$ just in the newer one, and $N_b = \mu_x(k) \cap \mu_{x'}(k')$ in both.

Let $v_o = \bigotimes_{y \in N_o} \alpha_y(\sigma_x^y(k))$ be partial result of the older aggregation, only over values from N_o, and let $v_b = \bigotimes_{y \in N_b} \alpha_y(\sigma_x^y(k))$, only over N_b. Then, $\beta_x(k) = v_o \otimes v_b$. Likewise, $\beta_{x'}(k') = v_b' \otimes v_n'$ for $v_b' = \bigotimes_{y \in N_b} \alpha_y(\sigma_{x'}^y(k'))$ and $v_n' = \bigotimes_{y \in N_n} \alpha_y(\sigma_{x'}^y(k'))$.

What we need to prove can now be stated 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) $v_o \otimes v_b \leq v_n'$. Specif cally, we prove the following chain of inequalities:

$$v_o \otimes v_b \overset{(i)}{=} v_o \otimes v_b \otimes v_b \overset{(ii)}{\leq} v_o \otimes v_b \otimes v_b' \overset{(iii)}{\leq} v_n' \otimes v_b' \,. \quad (44)$$

Here, (i) follows from idempotence of $\otimes$. Then, (ii) and (iii) follow from the monotonicity of $\otimes$ combined with (a) or (b), respectively.

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 $\otimes$ being a lower bound, and then $v_b \leq v_b'$ follows from (a). Finally, if only $N_o = \emptyset$, then (b) reduces to $v_b \leq v_n'$ as a special case, but the reasoning behind it remains the same as below.

Part (a) $[v_b \leq v_b']$. Since $k \leq k'$, and β is in-order, $\sigma_x^y(k) \leq \sigma_{x'}^y(k')$, and since α is weakly monotonic, $\forall_{y \in N_b}\, \alpha_y(\sigma_x^y(k)) \leq \alpha_y(\sigma_{x'}^y(k'))$. Since $\otimes$ is monotonic, we can aggregate all of these inequalities across $y \in N_b$. After simplifying, this yields $v_b \leq v_b'$.

Part (b) $[v_o \otimes v_b \leq v_n']$. 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 $\bigotimes_{z \in N_p^y} \alpha_z(\sigma_x^z(k)) \leq \alpha_y(\sigma_{x'}^y(k'))$. Aggregating all inequalities for $y \in N_n$ yields v_n' on the right side. Now, since $\otimes$ is idempotent, the left side is aggregation over $N_p = \bigcup_{y \in N_n} N_p^y$. Since $N_p \subseteq \mu_x(k)$ and $\otimes$ is a lower bound, excluding values from $\mu_x(k) \setminus N_p$ could only have made the result larger; hence, the left side is no smaller than $v_o \otimes v_b$. We conclude that $v_o \otimes v_b \leq v_n'$. □

C. PROOF OF THEOREM 2.3

PROOF. Take any two values $\beta_x(k), \beta_{x'}(k')$ f owing in β such that $k \leq k'$; we need to show that $\beta_x(k) \leq \beta_{x'}(k')$. By def nition, $\beta_x(k) = \alpha_{\mu_x(k)}(\sigma_x(k))$ and $\beta_{x'}(k') = \alpha_{\mu_{x'}(k')}(\sigma_{x'}(k'))$. Since α is monotonic, it suff ces to show that $\sigma_x(k) \leq \sigma_{x'}(k')$ to get the desired result. The latter trivially holds because β is in-order. □

D. PROOF OF THEOREM 2.4

PROOF. Let $\beta_x(k)$ be a value appearing in some sink $\beta \in H$. We construct a value tree with $\beta_x(k)$ at the root, in which each node has f nitely many children, the value in each node is an aggregation of values in all its children, and the hierarchy ref ects 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' is not a source. Equation (21) yields $\beta_{x'}(k') = \bigotimes_{y \in \mu_{x'}(k')} \alpha_y(\sigma_{x'}^y(k'))$; we add a child for each $y \in \mu_{x'}(k')$ and put value $\alpha_y(\sigma_{x'}^y(k'))$ in it. Indeed, the parent is an aggregate of all its children, by def nition 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 inf nite descending path in the tree, which would yield an inf nite descending chain of f ows, and this is impossible since the order on H is well-founded. Knowing that the tree is f nite and all leaves are sources, by associativity of $\otimes$ we can represent $\beta_x(k)$ as a f nite aggregation of values in sources. This concludes the f rst part of the proof.

Now, take any pair of values $\beta_x(k), \beta_{x'}(k')$, appearing in events e, e', and let t be the later of the times at which e, e' appear. Let H_t be a network obtained by truncating every f ow in H at time t. Now, since aggregation is always performed on past values, H_t remains well-def ned, and all our assumptions still hold. Only f nitely many aggregations could happen in a f nite time because we have assumed that $\mathcal{T}$ is isomorphic with $\mathbb{N}$. Each involves f nitely many nodes. For each such aggregation we build a f nite tree as explained above; then we truncate H_t, to leave only f ows that appear in the construction of those trees. The resulting network H_t' is f nite, so we can apply Theorem 2.2 inductively, starting from sources, and working towards β. Eventually, we obtain $\beta_x(k) \leq \beta_{x'}(k')$. □