Delegated, Sharded, Multicast Tree for Paranoid Stateful Lambdas

Marcus Plutowski
Willis Wang
Vivek Bharadwaj

Abstract We define a scalable multicast protocol for the Paranoid Stateful Lambda System that enables scalable traffic with a large number of PSL participants. Our multicast protocol dynamically builds a tree containing intermediate routers and clients with secure enclaves at the leaves, so that multicasts require only a small constant number of message routed per node. The topology reduces overall and per-router network congestion, Our protocol is simple and fault-tolerant, with nodes contacting a central coordinator to join and dropping off by simply refusing to maintain heartbeat connections. Our protocol also enables clients to advertise specific keys of interest to routers; as a result, nodes can avoid multicasting to all parts of the tree. We integrate our tree into the PSL codebase and verified correctness and load-balancing properties using a random sequence of tree joins and drop-offs. We then use a simulator written in Python to benchmark the effects of sharding the tree according to keys of interest.

1 Introduction and Related work

Edge computing is a distributed computing paradigm which involves the execution of application code on computing platforms "on the edge", often located close to users, mobile devices, and sensors. By bringing applications closer to users, edge computing enables access to simultaneously low-latency and high-bandwidth compute resources, increases data security and privacy, and allows the user to take back control of their data (Mor, 2020).

Though there are many benefits to running applications on the edge, doing so introduces new challenges which must be addressed by the underlying software infrastructure. Distributing computation to more but smaller devices puts greater strain on less powerful network hardware, while many applications demand high levels of interconnectivity that would impair attempts to fragment the network into smaller, more manageable chunks. Most challenging is the prospect of geographic scaling, where edge devices are not located within a single room or even city, but across whole continents — even the entire globe. Therefore, it is necessary for any such architecture to be able to consistently support high rates of communication between a large number of servers, without relying on high-speed routing hardware or high-bandwidth physical links; and in order to scale to the scopes envisioned by upper-layer projects in the field, it must be able to avoid producing bottle-necks either due to network packet congestion or due to dependencies on limited processing power.

1.1 Global Data Plane

One such approach in bringing a federated architecture to edge resources is through the *Global Data Plane* (GDP), providing a secure and and unified system in accessing edge resources (Mor, 2020). To maintain security while without trusting the infrastructure provider, code is executed within secure enclaves, which prevent malicious actors from reading privileged information.

The Paranoid Stateful Lambda (PSL) system provides a key-value store implemented on top of the GDP, allowing for multiple simultaneous writes with an eventually consistent semantics. The current multicast implementation of PSL sends signed writes to every other secure enclave through unicasts to each other secure enclave. While it is extremely responsive, as the number of enclaves increase, the number of packets packets are flooded throughout the network regardless of network topology, and the sender facing the bottleneck of network load, ultimately resulting in this scheme being unscalable. In order to ensure security of messages as they pass through the network, the PSL system takes advantage of the Global Data Plane's data routing structure, the DataCapsule.

DataCapsules are a key component of the Global Data Plane, providing all agents with access to easy-to-use, cryptographically sealed data containers for the edge (Mor, 2020). DataCapsules provide a uniform interface for data and metadata storage, along with a mechanism by which any agent in the Global Data Plane can uniquely resolve names to data. Internally, each DataCapsule uses a

blockchain- like mechanism to track provenance and prevent tampering. In this way, DataCapsules are an ideal storage container for the edge. All messages passed along the tree are sealed within DataCapsules, with their internal data only accessible via the private keys stored within enclaves on the network; message routing and other metadata is wrapped around the capsules to allow routers access, as the routers do not run within the enclaves on each server.

In what follows, we present a multicast system for the PSL system designed to address scalability of communication within participants, targeting key-value store operations. The protocol constructs and maintains a multicast tree shared across all nodes in the same PSL instance. Nodes use metadata from tree nodes to determine the routing procedure when sending multicast messsages. Our protocol has the following traits:

1. *Dynamic Reorganization*: As PSLs are incrementally deployable, PSL worker nodes can freely join and leave the cluster with minimal downtime. We perform liveness detection of nodes in a cluster using heartbeat monitoring and dynamically rebalance the tree in account for a growing or shrinking number of nodes.

2. *Message Filtering*: Messages may be filtered by intermediate routers, with messages associated with a particular key sent only to a subset of enclave participants who care about the key.

The current version of the PSL executes a multicast as a unicast from a centralized server to all clients. With n clients participating in the key-value store, a single multicast produces traffic that scales as $O(n)$ at the server. Our protocol decreases the number of messages sent at any node to a small constant (we benchmark a binary tree, so 3 in our case), enabling scalable all-to-all communication.

We built our protocol and integrated it with the existing PSL stack, verifying correctness and the tree balancing procedure. We then used simulations to measure the usefulness of message filtering / sharding in the tree.

1.2 Requirements

In this section, we describe the considerations in evaluating the scalability and performance of our proposed multicast protocol:

1. Congestion: Reducing the number of message sent both overall, and through each node

2. Performance: Latency and throughput of multicast messages should be comparable to a fully-connected broadcast scheme.

3. Security: Messages should continue to respect the security and provenance of information, and the security of the enclaves should not be compromised in any way.

2 Multicast Tree Protocol

In this section, we describe the life cycle of tree joining, leaving, heartbeat monitoring, and rebalancing our multicast tree. We also discuss the protocol's fault tolerance. We implement the protocol with a state machine abstraction, described more fully in the next section.

Each node in the tree has a single parent and may have both routers and clients as direct children. Links in the tree are maintained as TCP connections with associated metadata about the relationship between two nodes. The tree structure has the advantage of permitting easy sharding and isolation of tree components who may not all care about the same key values.

2.1 Node Types

The protocol has three node types:

1. **Root Router**: Receives and processes signals from an external control server and is the root node of the multicast tree. To join the multicast tree, clients and routers send join requests to the root and receive a parent in response.

2. **Intermediate Router**: Do not contain a secure enclave for computation and route messages between clients and other routers. Each router has a single parent in the tree, up to a specified number of child routers, and a specified number of child clients. A router may serve clients directly even if it has child routers.

3. **Clients**: Leaves of the multicast tree with secure enclaves for computation. Each client has a single parent in the multicast tree.

The root router must be active before any nodes can join the multicast tree, but clients and routers can join in any order. In particular, clients can join the multicast tree before any routers do and attach themselves directly to the root router. During subsequent tree rebalancing phases, these clients will be shuttled underneath intermediate routers to reduce bandwidth strain on the root node.

2.2 Join Requests

A router or a client asks to join the multicast tree by sending a "join request" message to the root router. The request contains the address of the joining node and a flag indicating whether the node is an intermediate router or a client (leaf).

If a client asks to join the tree, the root router checks if it has any intermediate routers as children. If not, it makes the client a direct child by:

1. Establishing a socket to the joining node
2. Sending an “assign parent" message back to the joining node
3. Registering the node with the heartbeat monitor to monitor liveness.

Otherwise, if the router has any child routers, it selects the child router with the fewest total children (arbitrating randomly in case of ties) and passes down the join request. The child router recursively uses the same protocol until some node in the tree sends an “assign parent" message back to the joining node.

Router joins follow a similar recursive protocol, with the caveat that the tree attempts to fill out the top levels of the tree instead of the bottom levels. A joining router preferentially becomes the child of the highest existing router in the tree; if that router has reached a maximum count of router children (some small constant), it passes down the join request in an analagous fashion to client joins.

Fault Tolerance: If the joining node has not received an assign-parent message after a predefined period of (local) time, it re-sends the request to the root router. When each join request arrives, routers check whether they already contain the new node in their list of children to avoid duplicating them in their local child lists. It is still possible, however, that two different nodes may send “assign parent" messages to the joiner and locally register that node as a child.

We remedy the situation as follows: the joining node changes its parent based on the latest “assign parent" message that it receives, and it ignores messages from any node besides its own children. It also refuses to send heartbeats to anyone but its defined parent. False parents remove the node from their list of children when they detect the lack of a heartbeat signal, and the tree remains coherent.

2.3 Multicast Routing

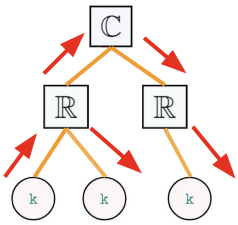

Figure 1: Client Executing a Multicast with Our Protocol.

Figure 1 illustrates a client node executing a multicast to all other nodes of the tree (assume no filtering based on keys for the moment). Messages may be injected into any node of our multicast tree, but typically originate at the root router and clients. A multicast message consists of a payload, the address of the last node in the tree who handled the message, and a flag indicating whether the message needs to travel up or down the tree. We include the flag for convenience at client nodes.

Multicast messages indicating updates to the distributed key value store begin at secure enclaves, which locally message the associated node on the common IO port directing them to pass the multicast message up. Clients pass the multicast message to routers, which in turn resend the message to all their neighboring nodes except for the one which provided the message to distribute. Routers avoid bouncing messages back and forth by using the “last handler address" included in each message. When multicast messages arrive at other clients, the flag within the message has been modified to note that they have been routed down the tree. They unpack the payload and send it to the secure enclave for decryption and processing. At each level in the tree, a router changes the “last handler" field before passing along the multicast message and also sets the routing direction flag according to the direction the message is traveling.

These features serve to reduce the number of unnecessary multicast forwards, but still send a large number of redundant packets in cases where some multicast messages are not relevant to subsections of the tree. This can be mitigated by the application of a Sharded Multicast Tree structure: by tracking which multicast messages it receives from each child connection, a router can determine what kinds of messages are in use by each worker below it. The symmetry between consuming and publishing multicast updates is essential here. This system can be efficiently implemented via the application of Bloom filters — while normally they would be difficult to use in this sort of situation, due to their not supporting subtraction operations, nodes that leave the tree will already necessarily have their Bloom filters thrown away, as each outgoing connection directly services one and only one router or worker.

Fault Tolerance: Multicasts may be interrupted by failing routers. Provided that routers other than the root fail, their children will rejoin the multicast tree, and the eventually consistent mechanics of the PSL design ensure long-term correctness of the key-value store.

2.4 Heartbeat Monitoring

To detect failures in the tree, each node in the multicast tree periodically sends “heartbeat messages" its parents and children. Each heartbeat contains the node address of the sender, the total size of the subtree maintained by a node along with additional useful metadata; we discuss this in the section on key advertising.

Each time a heartbeat is received from a neighbor, a node records the timestamp of the latest received heart-

beat. Periodically, nodes sweep their list of children; if a heartbeat has not been received in an appropriate time interval, the node clears the child and its associated metadata. If a node loses the heartbeat connection with its parent, it sends a “join request" messages to the root router and ceases heartbeat communication with its old parent.

Fault Tolerance: A child who does not receive a heartbeat in time will rejoin the multicast tree; the original parent, which may have stalled, still contains the node among its list of children, leading to a “false parenting" situation similar to the join request protocol. The solution is the same: nodes ignore messages from all but their sole designated parent and children, while false parents clear away the node from their registered list of children when they stop receiving heartbeats.

Congestion cost: Once concern with adding a heartbeat to the protocol was that it would add significant traffic going through the often already-overstretched root nodes. Three factors mitigate this. Firstly, it induces far less traffic than a comparably effective ACK-based protocol would. Secondly, the traffic it induces is distributed evenly across the tree, meaning that it increases congestion by a proportionally *smaller* amount in the root nodes than in the less-trafficked internal or leaf nodes. Finally, the heartbeat frequency tends to be low enough that heartbeat messages are far outnumbered by standard multicast propagation messages; this can also be tuned to fit the particular needs of each application.

2.5 Tree Rebalancing

Over time, imbalances may accumulate in the multicast tree. We employ a simple strategy to keep the tree balanced: periodically, nodes sweep over their list of direct descendants to check for severe imbalances in the total number of children (obtained from the heartbeat messages). If a descendant has a subtree that is too large, a “cull" message is sent to that descendant. Each cull message contains a single number: the node count to cull.

Upon receiving a “cull" message, a node ceases heartbeat communication with the specified number of its client children and removes them from its client list and sends a “disconnect" message to that client. If a node has fewer direct client children than the cull message specifies, it passes down further “cull" messages to descendant routers.

Upon failing to receive heartbeats or receiving a “disconnect" message, culled clients nullify their parent pointer and rejoin the tree. The join protocol originating at the root restores the tree balance.

Fault Tolerance: Culled nodes join the tree when heartbeat communications cease. Failure to pass down “cull" messages does not affect the correctness of the distributed key-value store or the multicast tree. Any imbalance that is not corrected is remedied in the next rebalancing pass.

2.6 Advertising Keys of Interest

Every multicast message in the PSL has an “key of interest" field. The field publicly exposes either a multicast key or a hash of the key.

Secure enclaves may multicast an “advertise interest" key to all ancestor nodes in the tree indicating their interest in receiving updates. If a multicast message has a nonempty key of interest field, it will only be routed through links to nodes that an express an interest in it.

Each intermediate router maintains a list of keys that each of its children is interested in. Multicasts always proceed up the tree, but routers may decide against routing them down. Note that we can tolerate false positives, but not false negatives preventing messages from reaching parts of the tree. As a result, Bloom, cuckoo, and quotient filters are space-efficient data structures to represent the key sets. While our current implementation maintains the key sets as simple lists, future implementations will incorporate filters. We simulate key distributions and filtering in our experiments section using Python.

Filtering for probabilistic routing in distributed computing settings has previously been explored (see the attenuated bloom filter, for example, a component of Oceanstore (Rhea and Kubiatowicz, 2002)). Since the PSL must support additions and deletes from the query set of interest, however, a cuckoo or quotient filter will likely be more appropriat. A good filter candidate must also be efficiently serializable to allow clients to address interest in a large set of keys “en masse" without transmitting excessive quantities of bits.

3 State Machine Implementation

In the protocol described above, nodes must participate in several activities (heartbeat sending, multicast routing, tree rebalancing) concurrently. To prevent these activities from interfering with each other, we adopt a state machine abstraction for each node of the multicast tree.

The state of any node in the multicast tree consists of its parent in the tree, a list of immediate child clients, and a list of immediate child routers. The node tracks the last heartbeat received for each neighbor and the subtree sizes for all child routers. Each node runs the following hot loop:

1. A message arrives at a node. Interrupt timers are handled by having the interruptor send a special “interrupt" message to the node’s main socket, which simplifies the loop design.

2. The node enters the state machine transition function. The node state evolves; for example, the node may accept a new parent or child. Messages may be dispatched to the secure enclave, multicasts may be routed, or heartbeats sent / received.

3. The node prepares to accept a new incoming message.

The sole trigger to enter the transition function is the arrival of a new message from another node, an interrupt to monitor or send heartbeats, or the secure enclave. Only a single thread on each node runs the hot loop.

Accordingly, we eliminate race conditions in the code whereby two competing events simultaneously attempt to evolve the node state. We note, however, that different orderings of message arrivals causes the node state to evolve differently.

3.1 Message Multiplexing

The original prototype for the PSL handled different message types by establishing a unique socket for each node (e.g. the root of the tree has separate sockets to handle messages from the control server, multicast messages, and to send results back to the synchronization server. Extending this design would require a different socket for each message type, which becomes untenable when nodes attempt to locate the correct socket to route messages into.

Instead, we use Google Protobuf to define a single ControlMessage type that is either a join request, multicast message, heartbeat, cull, etc. Upon receipt of messages, nodes use the message type to determine the appropriate state transition. Table **??** lists the message types in our protocol.

4 Python Simulator Harness

The actual implementation of the protocol erewhile described was wholly in C++; however, properly instrumenting the software proved difficult due to the complexity of implementing dependency injection for the CPU enclaves, the underlying network (this in particular was due to the Bazel build system impeding the use of standard networking simulators like NS3), and the user workloads. Most of the key metrics for Multicast Tree are based on network throughput, congestion, and security rather than speed, and are thus substrate-independent: regardless of the underlying hardware, our algorithm should respond in the same way to identical inputs, up to the influence of random effects in e.g. its Bloom filters.

As such, in order to simplify the process of benchmarking system throughput, we created a ground-up simulator for Multicast Tree in Python, completely replicating our algorithm in a fully managed environment. [1]

While this simulator is not suitable for end-to-end timing tests, which still had to be performed using the original C++ codebase, benchmarks relating to throughput and congestion were performed using this system. The behavior of the physical network, the enclaves, and the user agent are all simulated; everything else is largely true to the actual implementation, and can thus be relied upon to provide accurate results.

[1] `https://github.com/Achierius/py-multicast-sim`

5 Remaining Faults and Threats

The goal of the PSL is to allow potentially untrusted actors to participate in secure, correct computation with potentially confidential data. In this section, we discuss faults that our protocol does not yet address and the threat model of the multicast tree. We give avenues to fix some of these faults, some of which would require significant implementation effort.

5.1 Failure of the Root Router

The failure of the root router halts all execution of the PSL. Direct descendants of the root repeatedly attempt to get a new parent but fail, halting execution.

A potential solution would be the establishment of backup root routers that could function in case of failure of the primary. Implementing such a backup system would likely require a distributed consensus protocol such as Paxos (Chandra et al., 2007) or Raft (Ongaro and Ousterhout), which enables the pool of candidate roots to select a leader and reelect a leader in case of failures.

5.2 Data Integrity at Clients

The addition of a multicast tree does not affect the integrity of the PSL computation, since multicast nodes can only access encrypted versions of the data. Any change in the data payload can be detected as a break in the hash pointer chain stored by each datacapsule.

On the other hand, sharding the tree requires exposing either keys of interest, or a hash of those keys. While hashing the keys provides some measure of security, an adversary can determine information about a key using its send / receive frequency. It is impossible to avoid leaking such information without decreasing the effectiveness of sharding (e.g. a shorter hash output would make frequency analysis more challenging, but would also trigger more false positives of interest among multicast tree nodes.

5.3 Subtree Isolation

The PSL Key-Value store is eventually consistent; it can tolerate dropped packets and delayed routing as long as all nodes can exchange information with each other at some point in time. The sharding in the PSL, however, potentially enables malicious routers to refuse messages with particular keys from being routed to parts of the multicast tree. A malicious router can effectively insulate a router from all transactions pertaining to particular keys.

6 PSL Integrated Results

6.1 Demo of Our Multicast Tree

Interested users can run a demo of our multicast tree by switching to the PSL branch

```
cs262a_multicast_demo
```

spinning up a docker container, and running the single line:

```
bazel run //src:hello_world_sgx_sim
    -- --mode=6
```

The command issues (in expectation) 35 random router joins client joins in sequence interleaved with each other. After all routers and clients join, nodes randomly drop out until only the root is left. As nodes join or drop out, the tree dynamically reorganizes itself. Since we are mostly interested in the balance and packet efficiency of our tree, we do not spin up a coordinator or synchronization server. We did not simulate more than 35 nodes; we found that an increasing number of concurrent threads within the Docker image caused heartbeat signals to experience inordinate delays, resulting in nodes becoming prematurely stale and having to rejoin.

7 Correctness and Tree Balance

We verified correctness of our protocol by simulating random joins and drop-outs from the tree. In our random simulations, we found that every node who lost a parent successfully found a new one after resending a single join request to the root router.

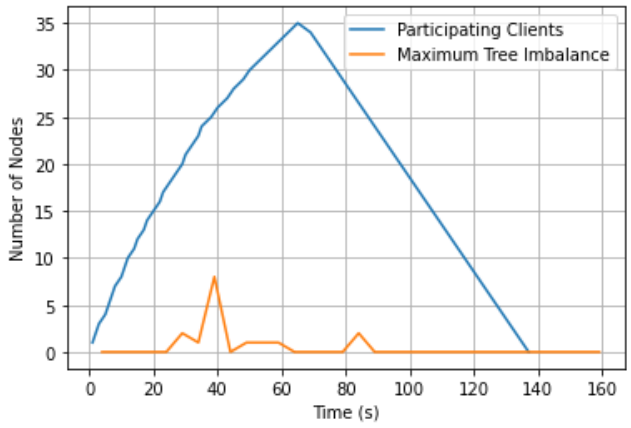

Figure 2: Maximum Tree Imbalance and Number of Participating Nodes over Time

Figure 2 shows the maximum tree load balance over 35 random client / router joins to the PSL and their subsequent random removals from the tree. In expectation, 10% of the nodes are routers. To produce the graph, we impose a binary tree structure. Accordingly, we define tree load balance as the maximum difference of total clients in the left and right subtrees of a router. Load imbalance temporarily spikes as router are added to the tree and nodes dynamically reorganize. Notice that at the beginning and end of the simulation, the tree imbalance is 0. This is because the root router services all clients at the beginning. At the end of the simulation, all clients are clustered under a single router that is parented by the root, again leading to a load balance of 0 by definition.

Figure 3 illustrates the average distance from the root of clients in the tree as a function of time. As new routers join, clients are pushed further away from the root of the tree (there is some delay between new routers joining and nodes reorganizing; 10 seconds are required for tree balancing signals), confirming that our protocol works as intended. After 100 seconds, all clients are three hops away from the root; when these intermediate routers are removed, all clients become children of the root. The gap in the red curve owes itself to lack of signal to routers as client nodes reorganize themselves by sending rejoin requests.

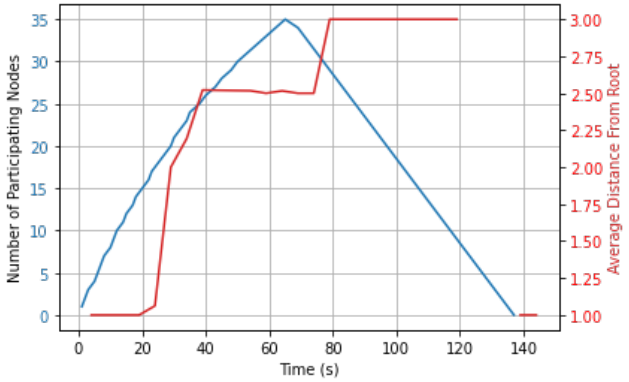

Figure 3: Average Client Distance from Root Number of Participating Clients over Time

Overall, our random simulations confirm that our protocol works as intended and supports multicast functionality identical to the original PSL codebase, with the advantage that the root router sends a small constant number of messages per multicast.

8 Sharding Python Simulations

In this section we give the results of simulating the key value store with our multicast tree. Simulation allows us to scale up to high node counts without the excess baggage associated with the full PSL.

8.1 Topology Comparison

For small networks, it is impossible to for a tree-based network topology to compete with a star or fully-connected pattern in terms of raw distribution speed; the indirection

of passing packets router-to-router from the tree's root to its leaves will always incur *some* cost.

As such, Multicast Tree was built with the intention of minimizing this (time-wise) performance penalty, rather than trying to improve time performance.

The most obvious downside of a star topology is that the star's central node naturally acts as a bottleneck as the number of network participants grows; several factors contribute to this:

- Hardware bandwidth limits: server network interface cards can only route a certain number of packets per unit time
- Software computation overhead: in cases where some amount of computation is performed per packet (e.g. choosing to which child a task will be assigned), the CPU poses an obvious cap
- Physical proximity: if the network spans a large geographical area, requiring that every packet route to and from a central arbiter (possibly on another continent) is extremely inefficient

Some of these factors (esp. #1, #2) can be mitigated with more expensive server hardware, but that is not something that can be assumed to be present in an arbitrary edge-computing problem context, and moreover even top-tier hardware will far exceed its limits at continental and global scales.

As such, reducing network congestion is by far the most important metric of success for the Multicast Tree. In order for it to feasibly scale, traffic across bottleneck routers should scale with $O(n \log n)$ in the amount of worker devices — and more preferably, scale with $O(\log n)$.

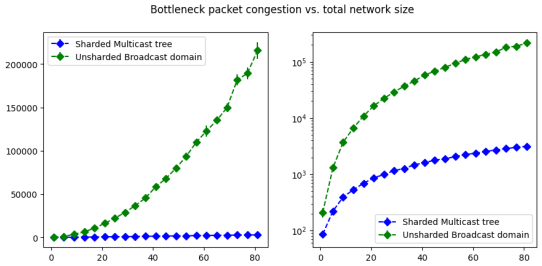

Figure 4: Congestion vs. Network Size

Figure 4 shows that the multicast tree does, as expected, put a significantly lower routing burden on the central router in the network. In the broadcast domain, the root router has to handle an amount of packets quadratic in the number of participant workers, while the throughput induced by the multicast tree only grows linearly. Our analysis, however, has shown that the root node is not the only possible bottleneck. The approximate member query filter-based sharding approach means that if a child router in the tree is allocated tasks with an aggregate working set with cardinality close to that of the multicast domain as a whole, its parent routers will forward it the vast majority of multicast updates that they receive. This results in internal routers becoming liable to suffer just as much, if not more, packet congestion than nodes near to the root of the tree. Figure 5 shows that standard load balancing (i.e. distributing worker nodes equitably across child subtrees) largely brings this effect within acceptable limits:

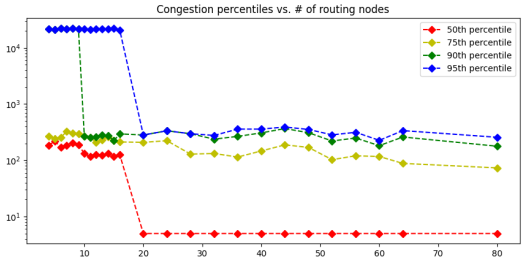

Figure 5: Congestion vs. Router Allocation in Fixed-Size Network

The choice of using a fixed-size network in Figure 5 was made to better demonstrates the effect of a deeper tree on routing congestion, as the effect would otherwise be obscured by the scaling of the network as a whole.

It is important to note that while this is indeed a limitation of the current approach, it is not a flaw: the number of multicast packets received by each such internal routing node is strictly lower than if the sharding were not in place, as the sharding protocol can only prevent packets from being sent that would otherwise have been sent anyways. See the future works section for further discussion of strategies for coping with this effect.

8.2 Impact of Sharding

We had at first considered utilizing one of the many off-the-shelf implementations of a multicast tree to address the PSL routing question; however, generic implementations all suffered from severe congestion at the root node, making them unusable for the edge-computing environment where Multicast Tree[] would be deployed. Sharding the multicast tree, as described above, proved effective at improving performance. The impact was such that it is not unreasonable to assume that this sharding, rather than the multicast tree, is behind the bulk of the improvement: in which case, it stands to reason that a simpler toplogy, without the structural overhead of a tree but utilizing sharding as done here, would be just as effective at reducing congestion in the root of the network and thus improving scalability overall.

It is as such important to separate the impact of sharding, as implemented via approximate member query filters maintained by each router in the multicast tree, from that the multicast tree eo ipso. One might imagine a star or

mesh topology with a similar sharding scheme in place: each (or the) router would maintain a Bloom filter for each outgoing connection, filtering outgoing multicast updates packets according to their keys' inclusion/exclusion in the filter corresponding to each link.

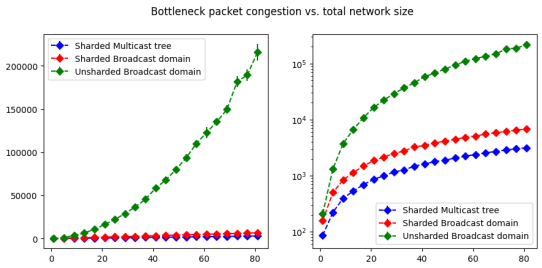

Figure 6: Congestion of Sharded Broadcast domain

Simulation of this strategy shows a remarkable degree of success, as demonstrated in Figure 6: while not matching the true multicast tree in terms of congestion reduction, it grows asymptotically at the same rate, at least within the range of scales the simulator is able to handle. However, while this approach seems promising in the simulator, it is not feasible in practice. The problem is that sharding requires a single approximate member query filter per network connection: as the size of the network grows, it is likely that more keys will be introduced into the tree, requiring larger and larger structures to maintain the same false-positive rates, even as more and more bloom filters are required to handle the additional connections being made. While the simulator does not place bounds on the in-memory size of the filters used, real-world devices certainly will. Table 1 shows how the size of a standard Bloom filter must scale in this situation.

# Keys in Filter	False Positive Rate	Filter Size
1,000	0.1%	2KiB
10,000	0.1%	18KiB
100,000	0.1%	175KiB
1,000,000	0.1%	2MiB

Table 1: Bloom filter size vs. # Keys in filter.

As noted, scaling in the number of links in the network is almost certainly going to come with scaling in the number of keys in each filter: the larger the problems being solved, the more need for larger networks to solve them, and the broader a region affected by a routine, the more variables it is likely to need to track. Storage space would thus be consumed quadratically in the scale of the network; a hypothetical global-scale multicast tree would require system memory capacities on the order of exabytes. To add insult to injury, many kinds of approximate member query filters cannot be scaled dynamically at runtime, but must instead be rebuilt from scratch each time.

In essence, in addition to its standard role as network-routing-mediator, the multicast tree serves to shard what would otherwise be a single, unmanageable approximate member query filter across the available hosts of the multicast domain.

8.3 Effect of User Programs on Performance

Thanks to the sharding approach described above, the behavior of user programs can have significant impacts on the performance of the Multicast Tree. In our canonical use-case — Paranoid Stateful Lambdas running on top of the Global Data Plane — this is because the multicast messages sent serve to maintain a distributed cache shared between the worker nodes; however, this effect persists in any situation which maintains distributed resources that are regularly accessed by worker nodes in a working-set pattern. If nodes in distant parts of the multicast tree share keys between their two working sets, then updates to those keys will require multicast messages to be routed all the way up to their common ancestor and back down again: if sufficiently unrelated, this may include propagation all the way up to the root of the tree, essentially entirely undoing the effect of sharding on the tree. Worse, the magnitude of this effect is correlated with the scale of the tree: as more worker nodes are added to handle larger and larger workloads, the probability of keys being present in the caches of workers connected to one another only by their shared connection to the root of the tree approaches 1.

As this effect compounds, the root invariably becomes heavily bottlenecked, posing a hard cap to system scale; avoiding this effect (and maintaining the viability of tree-sharding) is thus crucial.

8.3.1 Workloads compared

Figure 7 shows a height-3 balanced binary multicast tree, which we will use as a toy example for this purpose.

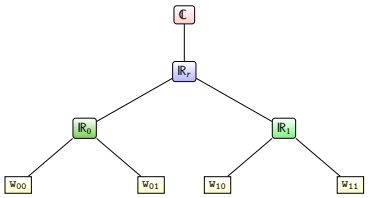

Figure 7: Balanced binary multicast tree

The simulator specifies each user-program by three parameters:

- The working-set of memcache keys used by the program
- The λ for the exponential distribution governing memcache write intervals

- The average number of memcache writes made per program execution

A "workload" for a given PSL instance can thus be described as a list of such user-programs. Structurally simpler workloads can be concisely described in terms of their independent working-sets: we will use $W(S \times K)$ to denote a workload consisting of S mutually-exclusive working sets with an average of K keys per set. This model, while too simple to be of use for more general benchmarks, is useful for illustrating the effect that an ideal distribution of cache access patterns has on the congestion of the Multicast Tree.

To demonstrate the key-separation effect, we selected four workloads of equal average intensity (both in terms of average execution duration and in the number of cache accesses made per execution): $W(4 \times 2)$, $W(2 \times 2)$, $W(1 \times 2)$, and $W(1 \times 1)$. Figure 8 shows how the number of packets processed by routers at different congestion percentiles changes as the workload of the system varies.

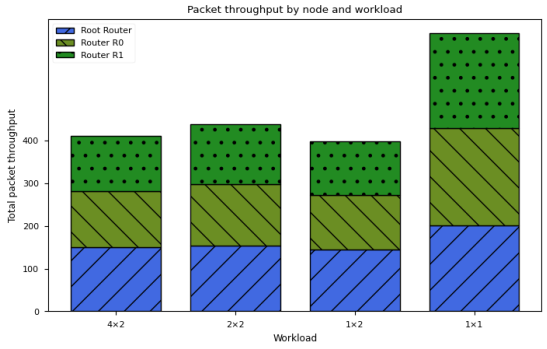

Figure 8: Avg. congestion of $\mathbb{R}_r$, $\mathbb{R}_0$, & $\mathbb{R}_1$ per workload (Static worker allocation)

8.3.2 Dynamic worker reallocation

The heartbeat signals sent by each router in the tree make it possible to introspect on the relative distribution of working-sets among nodes in the tree; by dynamically reassigning workers according to the keys for which they publish multicast messages, the coordinator can progressively reduce the number of keys whose multicast domains span the entire tree. The simplest approach is to inspect all multicast messages that cross the root (or other high-level tree nodes) and attempt to greedily reassign nodes to one side or another in order. This approach was able to reduce both total and root-level congestion for most workloads by around 20-25%, as demonstrated in Figure 9.

8.3.3 Pathological workloads

Consider the congestion induced by workload $W(1 \times 1)$ in the scenarios above: not only is it the worst-performing workload by far under both regimes, it saw little benefit

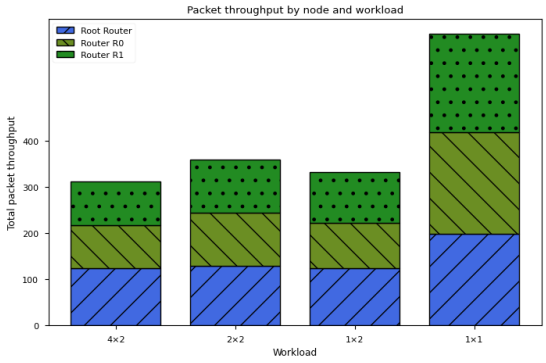

Figure 9: Avg. congestion of $\mathbb{R}_r$, $\mathbb{R}_0$, & $\mathbb{R}_1$ per workload (Cache-aware worker reallocation)

from the more advanced worker reallocation strategy. This is essentially the worst-case scenario for user workloads, as it represents a case where every write made by any worker in the tree must be published across the entire multicast domain, resulting in massive congestion at both the root and interior routers. Worker reallocation has no effect because each worker has an identical working set: there is no way to balance the workers across the tree without either overloading certain routers or requiring whole-tree multicast forwards for every memcache write. This scenario, while extreme, is certainly of concern, and many real-world workloads mimic it — linear algebra operations, for example, generally involve the traversal of large sets of data, where each key is accessed a small number of times by each parallel executor of the task. As such, this possibility demands further and future consideration.

9 Future Work

9.0.1 Scaling and Filtering

System resource constraints prevented us from scaling our PSL-integrated benchmark past 35 nodes. Future work might include running our multicast tree on physical hardware such as NUCs, or on a platform that can avoid contention.

Our filtering implementation currently does not integrate space efficient data structures, such as bloom / quotient filters. Rather, keys of interest are maintained as simple lists that we search, append to, and remove from. Integrating filters is relatively simple; in addition, sending serialized versions of the filters along with join requests would allow clients to advertise “en masse" a set of keys that they are interested in.

9.0.2 Alternative approximate member query filters

The most straightforward route to improvement, especially with regards to per-node routing performance, is replac-

ing the Bloom filters currently used for managing multicast packet filtering. The biggest issue with the use of standard Bloom filters is that they do not support set subtraction, and must instead be regenerated from scratch to remove keys from their internal set. This does not cause any problems with nodes joining/leaving the network: each outgoing connection has its own Bloom filter, which is necessarily discarded when the object on the other side of the connection crashes or disconnects.

Rather, set subtraction would enable the ability to *phase out* keys from the filter: as-is, long-lived connections steadily acquire a larger and larger working set of Bloom filter entries in the relevant parent router node, meaning that fewer and fewer multicast messages will be able to be discarded by the parent. This is the case even when those entries are no longer part of the child's actual working set. By periodically randomly removing keys from the filter, internal tree routers would be able to probabilistically track the actual working set of their children, mitigating the build-up effect described above.

Multiple other approximate member query filters support this operation; in particular, we investigated the use of Quotient filters, but were unable to polish the solution before submission. Nevertheless, we believe that this is the most promising approach: the option to optionally delete keys fits in quite well, and more importantly they can be merged easily without having to recompute keys, allowing for significant improvements over the current Bloom filter-based approach from a structural level.

9.0.3 Handling pathological workloads

The largest problem with the current implementation is its inability to scale with certain user workloads. As described above, hypothetical workloads such as $W(1 \times 1)$ force the tree to broadcast every multicast update to every node in the tree, causing severe congestion across nodes higher in the tree. While we have not had much success in our attempts to mitigate this thus far, we can broadly suggest two possible routes for further development.

Firstly, one could consider this a core trait of the system, and notify users of the system such that they introduce such pathological workloads at their own peril. This is not an unfounded approach: fundamentally, asking for a network computing on the edge to be able to handle routing of a single packet to hundreds of thousands of subscribers via a tree structure is not entirely realistic, and as such treating such a request as more-or-less equivalent to a denial-of-service attach is not out of the question. This does, however, pose some big issues. First of all, it is not immediately clear what the boundary between 'reasonable' and 'unreasonable' workloads is, or how a user would be able to determine on which side of that line their workload falls. Even beyond that, it will often be difficult for users to be able to predict the behavior of their workload at all: if the dispatch of keys is dependent on computations within the workload, this question is formally undecideable.

The alternative would be to attempt to alleviate these conditions within the structure of the tree itself. This is the approach that we have taken thus far, to little avail: generally, attempts to fix the routing have either compromised the eventually-consistent semantics of the system or the tree structure itself. The latter approach is the more palatable of the two, but while it is easy enough for the coordinator to decide to inform workers to send certain multicast messages to each-other directly (bypassing the tree structure) it remains to be seen how individual worker or router nodes would be able to store enough routing information to do so, or whether this would even solve the problem in cases where the multicast tree adheres to the actual underlying physical network topology (and thus routing between those points would cause physical congestion anyways).

Bibliography

Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing - PODC '07*, pages 398–407, Portland, Oregon, USA. ACM Press.

Derczynski, Leon. 2016. *Automatically Ordering Events and Times in Text*. Springer.

Mor, Nitesh et al. 2019. *Global Data Plane: A Federated Vision for Secure Data in Edge Computing*. Ph.D. thesis, University of California, Berkeley.

Mor, Nitesh et al. 2020. Global data plane: A federated vision for secure data in edge computing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 27(2):1–26.

Ongaro, Diego and John Ousterhout. In Search of an Understandable Consensus Algorithm. page 18.

Rhea, S.C. and J. Kubiatowicz. 2002. Probabilistic location and routing. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1248–1257 vol.3. ISSN: 0743-166X.