Jasper: Scalable and Fair Multicast for Financial Exchanges in the Cloud

Muhammad Haseeb
New York University

Jinkun Geng
Stanford University

Ulysses Butler
New York University

Xiyu Hao
New York University

Daniel Duclos-Cavalcanti*
Technical University of Munich

Anirudh Sivaraman
New York University

ABSTRACT

Financial exchanges have recently shown an interest in migrating to the public cloud for scalability, elasticity, and cost savings. However, financial exchanges often have strict network requirements that can be difficult to meet on the cloud. Notably, market participants (MPs) trade based on market data about different activities in the market. Exchanges often use switch multicast to disseminate market data to MPs. However, if one MP receives market data earlier than another, that MP would have an unfair advantage. To prevent this, financial exchanges often equalize exchange-to-MP cable lengths to provide near-simultaneous reception of market data at MPs.

As a cloud tenant, however, building a fair multicast service is challenging because of the lack of switch support for multicast, high latency variance, and the lack of native mechanisms for simultaneous data delivery in the cloud. Jasper introduces a solution that creates an overlay multicast tree within a cloud region that minimizes latency and latency variations through hedging, leverages recent advancements in clock synchronization to achieve simultaneous delivery, and addresses various sources of latency through an optimized DPDK/eBPF implementation—while scaling to 1000+ emulated receivers. Jasper outperforms a prior system CloudEx [12] and a commercial multicast solution provided by Amazon Web Services [38].

1 INTRODUCTION

There has been a growing interest from both industry [15, 16, 29, 31] and academia [12] in migrating financial exchanges to the public cloud because of multiple benefits provided by the cloud, including scalability, robust infrastructure, flexible resource allocation, and potential cost savings [6]. However, migrating a financial exchange from on-prem clusters to the cloud poses several challenges.

One fundamental requirement for a typical financial exchange is a fair multicast service [12]. Such a service (e.g., NASDAQ's ITCH protocol [28]) is responsible for sending data about the state of the market (also called *market data*) to a large number of market participants (MPs). This data is used by MPs to make trading decisions, and High Frequency Trading (HFT) firms often compete on how quickly they can place trades based on this information. This data should be delivered fairly, meaning every MP should receive the data almost simultaneously, ensuring no MPs have unfair advantages over others. Furthermore, firms require consistently low latency from exchange to MPs for market data distribution so that MPs trade on the most up-to-date information. While a high-performance fair multicast service in on-prem clusters might be implemented using switch support and carefully engineered networks,[1] the situation is much less favorable in the public cloud, where the hardware (e.g., switch) support for multicast is not usually available to cloud tenants. The public cloud also exhibits higher and more varied latency than on-prem clusters. As a result, implementing such a multicast service for financial exchanges in the public cloud becomes challenging, especially for cloud tenants.

In general, a financial exchange imposes two major requirements on the multicast service it employs. (1) *Scalability:* The number of MPs usually ranges from 100s to 1000s [30]. Market information needs to be multicast to a large number of MPs while maintaining consistently low latency in data delivery. (2) *Fairness:* Market data should be delivered to MPs nearly simultaneously with low spatial (i.e., across receivers) latency variance. Otherwise, MPs who receive data earlier would be able to act on it faster, disadvantaging other MPs in the system.

We find prior systems fall short of these requirements (§9). IP multicast does not meet our requirements as the public cloud does not provide switch multicast for cloud tenants due to scalability issues [39]. Application layer multicast (ALM) [3, 20, 27] focuses on improving throughput or latency by finding alternative "detoured" paths in an overlay network of machines in the wide-area Internet. We discuss in §2.2 how the absence of triangular inequality violations for path latencies in a single cloud region substantially limits the benefits one may get from finding the best paths in an overlay network of VMs. Some recent projects address the same

*Work done while at New York University

[1]Some financial firms measure wire lengths [40] to achieve simultaneous delivery of market data to all MPs.

question of multicast fairness as we do, such as CloudEx [12] and Octopus [15]. CloudEx achieves fairness, albeit with a significant increase in end-to-end latency, and scales to less than 100 receivers. Octopus relies on hardware support from cloud providers, utilizing SmartNICs to establish fairness, and it has only been shown to work for less than 10 receivers.

To satisfy the requirements of scalability and fairness while achieving low latency in the public cloud, we develop Jasper, an overlay multicast service for cloud-hosted financial exchanges. To achieve low latency while scaling to a large number of receivers, Jasper builds a tree of proxies for multicast, instead of having the sender directly unicast its message to all receivers. Effectively, a tree reduces the serialization or transmission delay required to unicast multiple messages back-to-back, but adds additional hops in the sender-to-receiver path (§4). Using such a tree for broadcast is an established technique in Message Passing Interface (MPI) literature [7, 37]. In Jasper, we use these trees as a starting point, and develop new techniques to lower latency and latency variance and achieve fairness in data delivery while scaling to a large number of receivers.

First, we define a mechanism for finding an appropriate *structure of tree* i.e., depth D and breadth/fan-out F, that achieves low latency for a given number of receivers by minimizing the impact of message replication and transmission delays. We propose a heuristic for the selection of D and F and demonstrate how a more sophisticated technique does not provide a substantial improvement over our heuristic.

Second, we introduce *VM hedging* in Jasper, motivated by request hedging [33], to lower latency variance, counter latency spikes, and narrow down the delivery window of a multicast message (i.e., the difference in time between the first and last reception of a multicast message). The main idea in VM hedging is that each node in the proxy tree (i) receives message copies from different sources and, (ii) processes the earliest received message copy and forwards that to children nodes. We run a Monte Carlo simulation of our hedging technique to establish its effectiveness.

Third, for fairness in data delivery, Jasper further builds on VM hedging to shrink the delivery window for a multicast message. We employ a *scalable message "hold-and-release"* mechanism based on the recent CloudEx system [12]. In this mechanism, the clocks of the sender and all receivers are synchronized using a high-accuracy software clock synchronization algorithm [11]. The sender attaches a timestamp called a "deadline" with each message. Receivers only process the message at or after the associated deadline. This deadline is set to be a point in time in the future at which all receivers are likely to have received the message with high probability. To estimate these deadlines, the sender needs to continuously collect one-way delay measurements from itself to all the receivers, and we develop a scalable technique

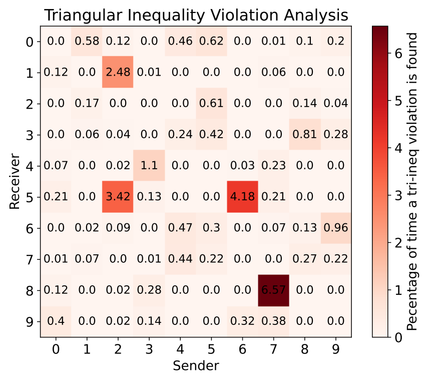


Figure 1: Violations of triangular inequality for path latencies are exceptionally rare within a cloud region.

for collecting such measurements, using the multicast tree in the reverse direction to aggregate measurement data from all of the receivers.

We evaluate Jasper against (i) direct unicast as a baseline, in which a sender directly sends a copy of the message to each receiver, (ii) an existing cloud-hosted financial exchange system, CloudEx [12], and (iii) a commercial cloud multicast solution provided by AWS [38]. We show that we scale to 1000+ emulated multicast receivers and achieve better latency and fairness than the baselines. We conduct further experiments to show the benefits of each individual technique: hedging, tree structures, and scalable fair delivery.

2 BACKGROUND

2.1 Migrating Financial Exchange to the Cloud: Requirements and Challenges

To meet their network performance requirements, financial firms currently utilize co-location facilities to deploy their financial exchange systems on highly engineered infrastructure. However, many of the techniques they employ to achieve this in the on-prem clusters are not available to cloud tenants, making it hard to deploy a scalable and high-performance exchange system in the cloud.

Lack of multicast support in the cloud. The *status quo* cloud environment does not provide switch support for message multicast. As a result, cloud systems implement the multicast functionality by directly unicasting a copy of a multicast message to each receiver [10, 12, 15, 16]. While the multiple-unicast approach works with a small number of receivers, it doesn't scale, since the message replication and transmission delay increases with each additional receiver. Indeed, some cloud providers, like AWS, provide their own proprietary multicast service (e.g., AWS Transit Gateways [38]) to tenants. This performs better than direct unicast but can only support ≈100 receivers (§8.1). Therefore, such multicast services given by the cloud provider still cannot suffice the demand of financial exchanges. To support many MPs, exchange operators need more performant multicast services that they can deploy by themselves as cloud tenants.

High latency variance in the cloud. In the public cloud, the latency between virtual machines (VMs) tends to have high variability and is prone to unpredictable latency spikes [16, 19, 26]. This makes it difficult to achieve the networking requirements of financial exchnages [16]. While on-prem clusters can mitigate or avoid latency fluctuations (e.g., using special hardware, and even using physical wires of equal length), public cloud tenants have very limited control over how packets are routed between VMs and cannot engineer the underlying network in the cloud. As a result, cloud tenants do not have the same ability as on-prem cluster users to tackle latency variance.

2.2 Cloud Latency Characterization

To gain a better understanding of the latency characteristics in the public cloud, we conduct a simple measurement study on AWS and GCP. Our measurement study reveals two main findings that motivate the design of Jasper.

Inter-VM latency exhibits significant temporal variance. As shown in Figure 2, the end-to-end delay between a pair of communicating VMs may show a significant variance. Latency spikes where the median latency increases by a couple of orders of magnitude have also been reported in the literature, but are rare [16, 26].

Violations of triangular inequality are rare in the cloud. Some networks, such as the wide-area network (WAN), may exhibit frequent and persistent patterns of triangular inequality violations (TIV) [23], meaning that the latency between one VM and another may be improved by routing the traffic through a third VM. It is therefore beneficial to find optimal "detoured" network paths in WANs when creating overlay multicast trees for low-latency communication [3, 4, 9, 18, 46]. However, in the single-region cloud settings we target, the network shows different characteristics from WANs and TIVs rarely occur. We have repeatedly conducted measurement experiments in both AWS and GCP. For each trial, we set up 10 VMs and measured the inter-VM one-way delay for two and a half hours. For each 1-second window, we find TIVs where latency from one VM to the other could have been improved by routing through another VM.[2] Our measurement study in both AWS (Figure 1) and GCP (Figure 12 in appendix) suggest very few TIV cases exist.[3]

[2]We may over-count TIVs as we only look at the granularity of 1 second.
[3]We look for TIVs where the latency between two VMs improves by at least 10 microseconds by going through a third VM.

Our motivation. Because of the absence of TIV cases in a cloud region, searching for the best paths in the network (just as what ALM does) no longer brings significant benefits. Instead, we shift our focus towards reducing the latencies incurred at *hosts*: we realize that the transmission (and replication) delay of multiple message copies becomes the primary bottleneck as the number of multicast receivers grows (details in §4). Therefore, a desirable approach to improving multicast performance in *a single-region cloud* should aim to reduce the delays incurred at the VMs when a sender is multicasting messages to multiple receivers.

3 JASPER DESIGN GOALS

3.1 Primary Goals

Goal 1: Support a large number of receivers. Typically in financial exchange systems, one or more **C**entral **E**xchange **S**ervers (CESs) multicast market data to a large number of MPs, which may be in the order of 100s to 1000s [24, 30]. Therefore, Jasper should be able to scale to handle thousands of receivers without causing bottlenecks at the sender.

Goal 2: Achieve consistently low latency. This implies two targets for Jasper: (1) the end-to-end latency of the multicast should be low so that the MPs are timely notified of the fresh information of the market; (2) the temporal variance of the latency should be minimal as the latency spikes may have adverse impacts on the system, e.g., lost trade opportunities, unfairness in data dissemination.

Goal 3: Ensure fair delivery. Fair delivery of market data to MPs is an important target [12, 15]. Achieving fair delivery requires the financial exchange system to make every MP receive the same market data simultaneously. Therefore, for each multicast message, Jasper needs to achieve minimal spatial latency variance (i.e., across receivers) so that each receiver may access the message at almost the same time.

Besides the primary goals, high message throughput is also an objective of financial exchange systems.

3.2 System Overview

Jasper borrows the idea of trees for scaling communication to a large number of receivers while retaining low latency [7, 37] (Figure 3). In Jasper, different market data streams (e.g., different ticker symbols) are assigned to independent trees that might share common receivers—allowing us to scale throughput via sharding (§7). Here, we focus on a single tree, where each tree node only has a small number of children that they are responsible for directly sending data to.

Tuning the tree structure (§4): The structure of a proxy tree (depth D and fan-out F) is tuned to minimize latency. Existing cloud-based exchanges [12, 15, 16] implement multicast using the direct unicast approach. This may be considered

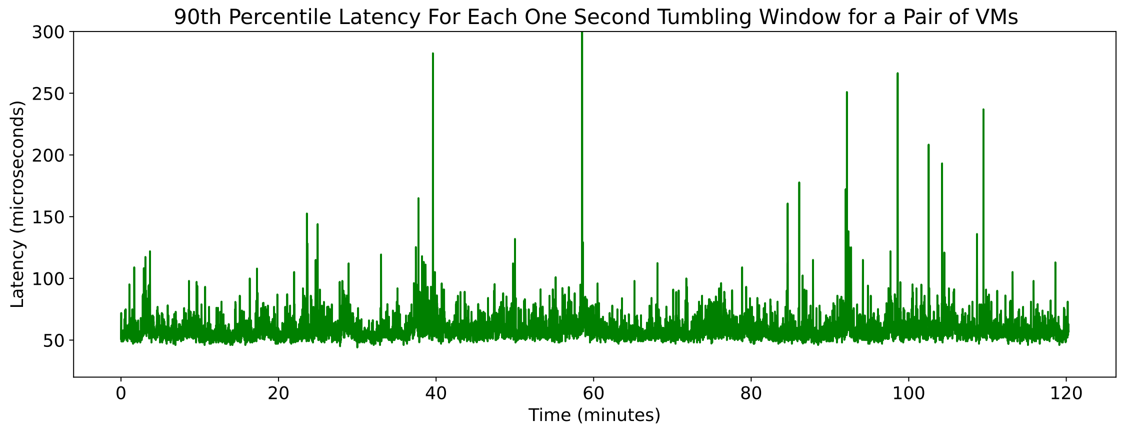


Figure 2: Latency between a pair of VMs may show significant temporal variance.

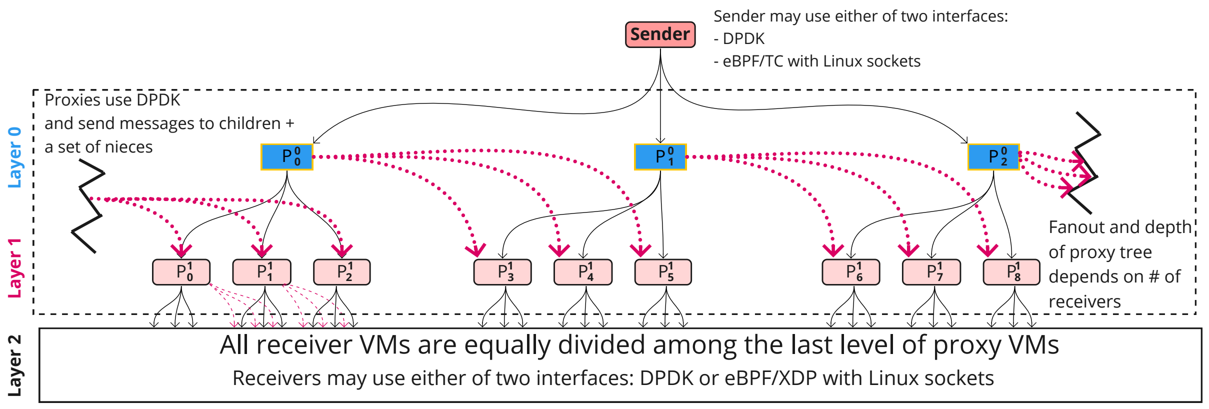

Sender
Sender may use either of two interfaces:
- DPDK
- eBPF/TC with Linux sockets
Proxies use DPDK and send messages to children + a set of nieces
Layer 0
Layer 1
Layer 2
P^0_0
P^0_1
P^0_2
P^1_0
P^1_1
P^1_2
P^1_3
P^1_4
P^1_5
P^1_6
P^1_7
P^1_8
Fanout and depth of proxy tree depends on # of receivers
All receiver VMs are equally divided among the last level of proxy VMs
Receivers may use either of two interfaces: DPDK or eBPF/XDP with Linux sockets

Figure 3: Several such trees terminate on the same receivers. Trading symbols are sharded across all the trees. In a tree, each node receives messages from 2 or more varied sources (not all edges are shown in the diagram). Dotted edges represent hedging (§5), which can effectively lower the latency variance.

a special case of a tree where D is 1 and F is N. We show that there is value in increasing the D and decreasing the F as the number of receivers (N) grows (***Goal 1***). We provide a simple heuristic for tuning D and F which provides good performance. We discuss how a more sophisticated mechanism for tuning does not outperform our proposed heuristic because of the high variation in latency and performance of VMs in the public cloud.

VM hedging (§5): For achieving consistently low latency and decreasing the latency variance spatially, Jasper introduces a technique called VM hedging, motivated by request hedging [8, 34, 42]. In VM hedging, each VM in the proxy tree receives messages from two or more different sources (i.e., parent and one or more aunts) where the path lengths for all messages are the same. A VM processes and forwards the message copy to children that is received earliest and discards the rest. VM hedging reduces the impact of latency fluctuations, yields much smaller latency variance, and narrows down the window of time in which all the multicast receivers receive a multicast message (***Goal 2*** and ***Goal 3***).

Scalable simultaneous delivery (§6): Simultaneous delivery is achieved by a scalable message *hold-and-release* mechanism. This mechanism was first introduced in [12] and works effectively for O(10) receivers. Our scalable version of this mechanism employs the tree in the reverse direction to collect and aggregate latency measurements required for the *hold-and-release* mechanism. In combination with VM hedging to reduce latency variance, we can achieve simultaneous delivery for 1000s of emulated multicast receivers (***Goal 3***).

Optimizations for throughput (§7): We engineer each proxy node in a tree to efficiently perform the task of receiving a message, replicating it, and sending it to children. We use

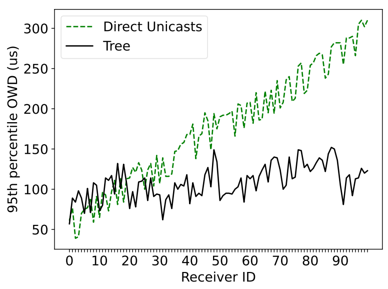


Figure 4: Without a tree, increasing number of receivers increases message latency for later receivers.

(a) N=10

D	F	OML
1	10	66
2	4	88

(b) N=100

D	F	OML
1	100	351
2	10	139
3	5	141

(c) N=1000

D	F	OML
2	32	282
3	10	217
4	6	226

Table 1: Median values of overall multicast latency (OML) for different depth (D) and fanout (F), for a given number of receivers (N). OML for a multicast message is defined as the max of one-way delays from the sender to all receivers.

a multi-threaded DPDK implementation with Zero-Copy message replication for high throughput and low latency.

Optimizations for usability (§7): Jasper implements 2 interfaces (for sender and receivers): (1) a DPDK interface, which uses DPDK on the senders, receivers, and proxies and (2) a socket interface, which uses eBPF/TC at the sender for message replication, and eBPF/XDP at the receivers for message de-duplication. Type (1) yields higher performance than type (2) but requires porting applications to use DPDK. Type (2), allows legacy socket-based applications to use Jasper without changing their network APIs. For type (2), we leverage eBPF to minimize the number of system calls when conducting multicast operations. Therefore, while type (2) is not as performant as type (1), it still yields higher performance than using sockets without eBPF to multicast the messages. Our eBPF implementation de-duplicates multiple hedged packets with a fixed length buffer (as required by eBPF) and using the limited programming primitives available in XDP.

4 TREES FOR SCALABLE AND LOW-LATENCY MULTICAST

For 1-to-N communication in the public cloud, a commonly used technique is that a sender replicates the messages and sends one copy to each receiver. When a sender transmits multiple copies of a message back-to-back, the latency of later messages increases due to the transmission delay incurred by the transmission of earlier messages. Figure 4 shows the latency of messages experienced by different receivers when sent using unicast and when employing a proxy tree (Figure 3) to minimize the effects of transmission delays to achieve lower end-to-end latency.

A proxy tree structure may be defined by 2 factors: the depth D and the fan-out factor F. D indicates the number of hops from the sender to each receiver. F indicates the number of message copies that the sender/proxy needs to send (via unicast) for each message to its downstream nodes. All the receivers are placed in the last layer of the tree, and they are evenly distributed among the proxies in the level above. Therefore, the last layer of proxies may have a different fan-out factor than F. For supporting N receivers, there are multiple <D, F> configurations. Some configurations have lower transmission delay because of lower F but they also have extra hops in the messages' paths because of high D and vice versa. For minimizing the end-to-end multicast latency, we need to find the best <D, F> configuration, given N.

Deciding D and F for multicast tree. To understand how the end-to-end latency varies as D and F grow, we conduct a series of experiments with various numbers of receivers ($N = 10, 100, 1000$). Table 1 displays the latencies for different configurations of <D, F> for a given number (N) of receivers. We see that the latency reaches a minimal point at different depths D for different values of N. At a small scale ($N = 10$), increasing D does not bring latency benefits. As the scale grows from $N = 10$ to $N = 100, 1000$, the benefits of increasing D while reducing F grow because the reduced message replication delay and transmission delay outweigh the added overhead of new hops in the path of messages. However, as D goes beyond a certain threshold (e.g., D grows larger than 3 when $N = 1000$), the latency does not improve and may worsen. Based on our experiments, we find fixing $F = 10$ usually leads to a desirable D to generate a multicast tree with low latency. We establish our heuristic rule to construct the multicast tree as follows: Given N receivers, we fix $F = 10$ and then derive $D = [log_{10}N]$ (round to the nearest integer). We also tested a more sophisticated alternative mechanism as described in [41], and did not observe a significant performance boost compared to our heuristic.

In our testing, we find that variance in VM performance in the cloud [36] makes it challenging to differentiate the "optimal" values of D and F from values picked using the aforementioned heuristic. We observe through repeated experimentation that minor changes in D and F do not show a significant performance difference with high statistical confidence. So it is sufficient to select D and F in the neighborhood of the optimal value, which is why our heuristic is effective. We further find that a unit increment in D comes at the added latency of $30 \pm 10\mu s$ and a unit increment in F adds $2.7 \pm 0.9\mu s$ per layer.[4] A tree constructed using linear models based on these unit increments performs comparably to the tree constructed using our heuristic of maintaining F to 10 and $D = \lceil log_{10}N \rceil$.

5 VM HEDGING

5.1 Hedging Design

Although the tree structure enables Jasper to provide scalable multicast, the latency variance becomes more significant and spikes occur more frequently as the tree grows larger. This not only impacts the latency of our system but also our ability to deliver messages fairly. It's been well established that the public cloud generally has more variable latency than on-prem solutions [33, 45]. This impact is amplified in a tree since a latency fluctuation in any node (or path to a node) increases the end-to-end latency of all the receivers whose transmission path includes that node.

To combat latency variance, we develop a *VM hedging* strategy in Jasper: we allow each proxy to receive multiple copies of a message (from the higher level of proxies), and the proxy only processes the copy that arrives first and ignores duplicates. To do this, we first describe a strawman design that leads us to our final design described next.

Strawman hedging design: F+H Technique. Our original hedging design is that each proxy node has H extra children along with its original F children, where H is the hedging factor. For example, in a multicast tree with $D = 3$ and $F = 3$ (Figure 3), if we configure $H = 1$, we will add 1 hedge node in Layer-0 and 3 hedge nodes in Layer-1. Each hedge node will assist the F original proxies at the same layer and does not have children of its own. A hedge node assists these F proxies by forwarding a message received from the hedge node's parent to the hedge node's F^2 nieces. In this way, a proxy node (other than the ones in the top level) will receive $1 + H$ copies of the same message: 1 copy from its parent, and the other H copies from the hedge nodes of the previous layer. Empirically we find that this technique significantly reduces latency variance. However, it has a distinct drawback: Whereas a proxy only needs to multicast messages to the F nodes, the hedge node needs to multicast the same message copies to F^2 nodes, creating a throughput bottleneck at the hedge node. We tried allowing a hedge node to only multicast to a subset of F nieces instead of all F^2 nieces. While this alleviates the throughput bottleneck, we no longer see the same decrease in latency variance. Therefore, we build on the lessons to design a new hedging scheme.

Final hedging design: SiblingHedging. Instead of adding extra hedge nodes in a tree, we let each proxy node also take on the responsibility of hedging. Specifically, each proxy node sends messages to the children of H of its siblings along with sending messages to children of its own. For example, in Figure 3, when hedging is not enabled (i.e., $H = 0$), proxy[5] P_3^1 only receives messages from P_1^0. As a result, P_3^1 may suffer from high latency if P_1^0 or the path from P_1^0 to P_3^1 encounters latency fluctuations. By using hedging, P_3^1 not only receives messages from P_1^0, but also receives the same messages from P_0^0 (if $H = 1$). With hedging, there are multiple paths from the sender to any P_i^j where $j > 0$, so the negative impact of latency variations can be effectively reduced. This technique demonstrates significant benefits that we evaluate in §8.2. It also does not introduce any significant bandwidth bottlenecks, unlike the $F + H$ hedging technique. Next, we present a random variable analysis to further understand SiblingHedging.

5.2 Hedging Analysis

When VM hedging is enabled, we may model the latency experienced by a receiver as follows.

We use function $\mathsf{L}(a, b)$ to represent the latency from node a to node b. We use S to represent the root node (i.e., the sender) in Figure 3, and P_i^j to represent the node i (i.e, a proxy or a receiver) in Layer j. Then, given a node P_i^n, the end-to-end latency from the root sender to this node can be recursively defined as the following random variable ($\mathbb{U}$ denotes uniform random distribution):

$$\mathsf{L}(S, P_i^n) = \min_{0 \le j < H} \left\{ \mathsf{L}(S, P^{n-1}_{(\lfloor i/F \rfloor - j)\% H}) + \mathsf{L}(P^{n-1}_{(\lfloor i/F \rfloor - j)\% H}, P_i^n) \right\}$$

$$\forall i, j : \mathsf{L}(S, P_i^0) \sim \mathbb{U} \text{ and } \mathsf{L}(P_i^{n-1}, P_j^n) \sim \mathbb{U}$$

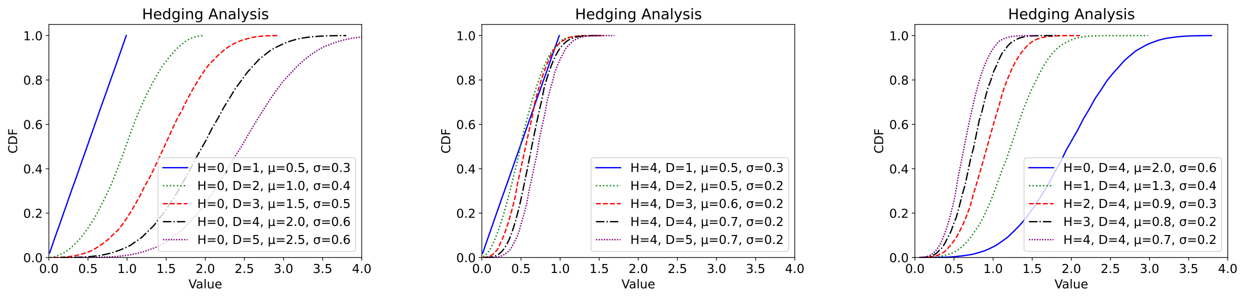
Achieving a low variance of $\mathsf{L}(S, P_i^n)$ would mean that the latency over time may not deviate much from the expected value, helping in achieving consistent latency over time. It also shows that different VMs (at the same level of the tree) in Jasper may not experience latency significantly different from each other, reducing the difference between the maximum and minimum latency experienced among all the receivers for a multicast message.

We run a Monte Carlo simulation of $\mathsf{L}(S, P_i^n)$ random variable with different values of H and D to understand its behavior. The simulation is run for 100k iterations. Based on the simulation results (Figure 5), we have three main takeaways.

No Hedging ≈ High Latency Variance: With no hedging, the depth of a tree and latency variance are directly correlated. Figure 5a plots the CDF for $\mathsf{L}(S, P_i^n)$ and shows mean value μ and standard deviation σ for different configurations of the multicast tree. As D grows larger, we see both μ and σ increase distinctly, indicating that just a proxy tree

[4]Using our high-performance implementation on a c5.2xlarge VM in AWS

[5]P_i^j represents i-th proxy in Layer j.



(a) With no hedging, the depth of a tree makes the latency and its variance worse.

(b) Hedging limits the impact of depth on the latency and its variance.

(c) For a fixed depth, increasing H improves the latency and its variance.

Figure 5: Analyzing VM Hedging. A Monte Carlo simulation with 100k iterations was used.

(i.e., no hedging) suffers from more latency variance when the tree scales up.

Hedging ≈ Low Latency Variance: Our hedging makes the correlation between D and σ become less significant. In Figure 5b, we can see the latency distribution becomes *narrow* (i.e., reduced σ) as H grows from 0 to higher values.

Low Overall Latency & Diminishing Returns on H: In Figure 5c, we fix D and keep increasing H. Figure 5c shows that hedging not only reduces the latency variance (i.e., the distribution becomes narrower), but it also helps to reduce the overall latency (i.e., the distribution moves leftwards). However, as H grows larger, the incremented performance gains diminish, and most performance improvement is obtained when H grows from 0 to 1. It shows that a small value of H (>0) is enough to reap the benefits of VM hedging. This is useful because a small H saves bandwidth.

6 SCALABLE SIMULTAENOUS DELIVERY

Financial trading needs to ensure the fairness of the competition. Fairness in data delivery [12, 15] means that the market data from the exchange server should be delivered to every MP at the same time so that an MP may not gain an advantage over the others during the competition. While there are also some recent works trying to alter the definition of fairness [16, 25], these variant definitions require more research before they are confidently adopted. Therefore, Jasper uses the original definition of fairness employed by on-premises financial exchanges. In short, in Jasper the fair delivery of data means simultaneous delivery of market data to all the multicast receivers.

Realizing perfect simultaneous data delivery to multiple receivers is theoretically unattainable [14]. Nevertheless, Jasper tries to empirically minimize the spatial (i.e., across receivers) variance of the latency of messages. Our hedging design (§5) has created favorable conditions to minimize the spatial variance. By using hedging, Jasper can achieve consistently low variance for each receiver over time, and the spatial latency variance across receivers is kept low. Beyond this, we employ a *hold-and-release* mechanism (by using synchronized clocks) to eliminate the residual spatial variance at the end hosts and enforce simultaneous delivery across receivers. The *hold-and-release* mechanism was introduced by CloudEx [12] but it does not scale well and leads to high end-to-end latency (§8.3). We describe a modified *hold-and-release* mechanism that helps us scale further, while maintaining a low end-to-end multicast latency.

Hold & Release mechanism. To implement the *hold-and-release* mechanism, Jasper leverages the accurate clock synchronization algorithm, Huygens [11], to synchronize the clocks among the sender and receivers. Receivers keep track of the one-way delay (OWD) of the messages received from the sender. Each receiver takes the 95th percentile of its OWD records at regular intervals and sends the results back to the sender using an all-reduce mechanism explained later. The sender calculates the maximum of these OWDs called Global OWD for messages using the gathered statistics: $\mathbf{OWD_G} = \max_i(\mathbf{OWD_i})$ where $\mathbf{OWD_i}$ is the OWD estimate reported by the i-th multicast receiver.

Once an $\mathbf{OWD_G}$ has been calculated by the multicast sender, it attaches deadlines to all outgoing messages. A deadline is calculated by adding $\mathbf{OWD_G}$ to the current timestamp when sending a message. Upon receiving a message, a receiver does not process it until the current time is equal to (or exceeds) the message's deadline. This mechanism leads to almost simultaneous delivery, modulo clock sync error. In §10, we discuss possible security mechanisms to ensure that a receiver (an MP) waits until its deadline to process a message, when it's in the MP's self interest not to wait.

Deadlines all-reduce. In the design of Jasper's *hold-and-release* mechanism, all the receivers have to send estimates of the OWD they experience back to the sender so that the sender can estimate the deadlines for the subsequent messages to multicast. If all the receivers send their estimated OWD directly to the sender, incast congestion occurs at the sender, leading to increased message drops.[6] To avoid the high volume of incast traffic, we reuse the multicast tree to aggregate the OWD estimates in an all-reduce manner. Specifically, each receiver periodically sends its OWD estimate to its parent proxy. As each proxy has a limited number of children, we do not risk in-cast congestion here. Each proxy (i) continuously receives estimates from its children; (ii) periodically takes the maximum of all the received estimates, ignoring some children OWDs if they have not yet sent in an estimate and; (iii) and sends the max value to its parent proxy. In this way, the sender at the root calculates deadlines for messages by only receiving the aggregated estimates from the first proxy layer instead of all the receivers.

7 OPTIMIZATIONS FOR HIGH THROUGHPUT AND USABILITY

7.1 Throughput

Decoupling DPDK Tx/Rx processing: We leverage the high-performance lockless queue [21] provided by DPDK to decouple the Tx/Rx processing logic. On each 8-core proxy VM in Jasper, we allocate one core (i.e., one polling thread) for Rx and 6 cores for Tx. The Rx thread keeps polling the virtual NIC to fetch the incoming messages and dispatch each message to one Tx thread via the lockless queue, which continues to replicate and forward the message.

Minimizing packet replication overheads: When a Tx thread is replicating the message, instead of creating F packets with each containing one complete copy of the message, we use a zero-copy message replication technique. For each packet, we remove the first few bytes that contain Ethernet and IP header and then invoke $rte\_pktmbuf\_clone()$ API to make several shallow copies of the packet, equal to the number of downstream nodes of a proxy. We allocate small buffers for new Ethernet and IP headers (from DPDK memory pools) and attach each pair of these buffers to one shallow copy created previously. Then we configure the headers properly (writing the appropriate destination addresses) and use $rte\_eth\_tx\_burst()$ API to send the packets out.

Parallelizing multiple multicast trees: To further improve the throughput, we borrow the sharding idea used by CloudEx [12]. Since each piece of market data is associated with one trading symbol (e.g., $MSFT, $AAPL, $AMD), we can employ multiple trees in parallel to multicast the market data associated with different symbols. In this way, Jasper's throughput scales horizontally by adding more multicast trees.

[6]We measure that more than 20% of the packets are dropped when 100 receivers attempt to send OWDs back to the sender simultaneously.

7.2 Usability

For the sake of high performance, Jasper implements the message processing logic by fully using DPDK. However, we also realize that many legacy applications are programmed based on the POSIX socket APIs, and porting them to the DPDK-based APIs is costly and challenging. So we implement a second interface with UNIX sockets at the sender and receivers.[7] To minimize the performance sacrifice, we leverage eBPF to reduce several overheads.

First, the sender, instead of sending F messages to Layer-0 proxies, sends one message using a UNIX socket which is then captured by an eBPF/TC hook. The hook replicates the message using $bpf\_clone\_redirect()$ and sends one copy to each of the F Layer-0 proxies. This eliminates the overhead of creating message copies in the userspace and sending each copy to the kernel.

Second, receivers install an eBPF/XDP hook which makes sure that the earliest copy of a multicast message is received and sent to the application. The later copies are dropped by the hook. Due to hedging, a receiver gets $H + 1$ copies of a multicast message where the earliest one is supposed to be accepted. The XDP hook serves that purpose. This optimization keeps duplicate messages from traversing the kernel stack and going to userspace. However, implementing a packet de-duplication mechanism is challenging in XDP as only fixed-length buffers are allowed and several operations (e.g., modulo (%)) are not permitted. So we devise an algorithm that works under the assumption that messages which are sent by the sender one second apart do not get re-ordered. Details of our de-duplication implementation are described in Appendix A.

8 EVALUATION

We answer the following questions.

(1) How does Jasper compare to the direct unicasts scheme and AWS Transit Gateway? §8.1
(2) How does VM hedging perform? (§8.2)
(3) How effectively does Jasper achieve simultaneous delivery? How does it compare to CloudEx? (§8.3)
(4) What is the impact of Jasper on trading algorithms? (§8.4)
(5) How scalable (i.e., number of receivers) is Jasper? (§8.5)
(6) How much throughput can a Jasper tree provide? (§8.6)
(7) How does multicast latency change as each component of Jasper is introduced? (§8.7)

Our experiments are performed on AWS with c5.2xlarge VMs. Unless mentioned elsewhere, we use 100 receivers and the sender's sending rate is 5K multicast messages per second. We only enable Jasper's simultaneous delivery in §8.3-§8.5. All the CDFs presented in this section show percentiles in the range [1, 99]. Each experiment case is run for 150 seconds.

[7]Proxies between sender and receivers are still implemented with DPDK.

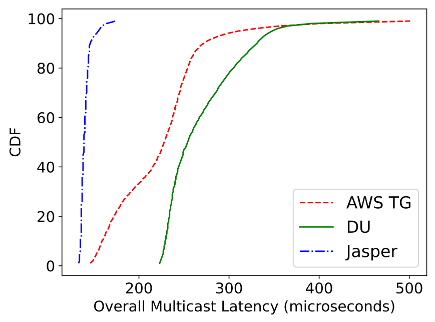


Figure 6: Jasper outperforms DU, and AWS TG in terms of achieving a lower OML.

One important limitation of our experiments is that we host 10 receivers per VM for $N = 1000$ and 20 receivers per VM for $N = 5000$. We are limited by VM quotas to test our hypotheses and expect to resolve this in the future. This leaves open the possibility that our results might worsen if we use 1 receiver per VM at high N.

Below are the common definitions used in the evaluation:

OML: When a message is multicast to multiple receivers, the Overall Multicast Latency (OML) refers to the latency experienced by the last receiver that receives the message.

Delivery Window: It is calculated as the difference between the OWD (one-way delay) experienced by the receiver who receives the message earliest and the OWD experienced by the receiver who receives the message latest.

8.1 Comparison with other techniques

We consider the following baselines and compare their multicast performance with Jasper.

• Direct Unicast (DU): When conducting multicast, the sender directly sends a copy of a multicast message to each receiver. We use `io-uring` to create a strong baseline as it reduces the latency by minimizing the overheads of syscalls.

• AWS Transit Gateway (AWS-TG): AWS-TG is the multicast approach provided by AWS [38]. AWS-TG makes the sender send the message to a gateway which then replicates and sends one copy to each receiver. The details of AWS-TG's implementation are proprietary. However, our measurement shows that AWS-TG can support at most 100 receivers per multicast group.

Figure 6 shows that Jasper distinctly outperforms DU and AWS-TG. The median latency for Jasper is $129\mu s$ where it is $228\mu s$ for AWS TG and $254\mu s$ for DU.

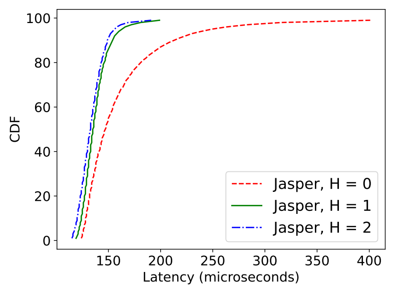
8.2 Hedging Evaluation

Reduced overall multicast latency. For every multicast message from the sender, we record message latency for every receiver and take the maximum among these N latency values as the *overall multicast latency* (OML) for a message. Figure 7a compares the OML CDFs under different hedging factors ($H = 0, 1, 2$). $H = 0$ represents the case when hedging is not enabled. Compared with $H = 0$, $H = 1$ yields distinct latency reduction as the CDF curve is shifted to the left. As we increase H from 1 to 2, the latency reduction becomes marginal. However, we show later that for a larger number of receivers, setting $H = 2$ yields better results than $H = 1$.

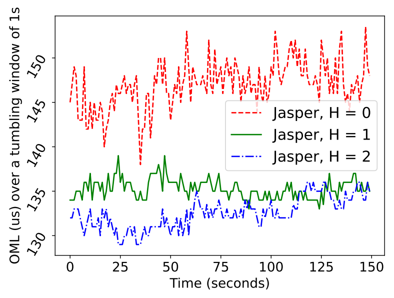
Reduced temporal latency variance. In Figure 7b, we calculate the median OML over a tumbling window of 5000 messages to study the temporal latency variance during the multicast. Figure 7b shows Jasper with hedging ($H = 1$) exhibits lower latency variance compared with the non-hedging scheme ($H = 0$).

Reduced spatial latency variance. Figure 7c compares the CDFs of delivery window size for Jasper with different hedging factors. We can see that hedging substantially reduces the spatial latency variance (i.e., it shrinks the delivery window of multicast messages). The 99th percentile delivery window size is ∼350 µs with no hedging, but is reduced to ∼150 µs when Jasper uses $H = 1$. However, as H grows from 1 to 2, the window size reduction, although it still exists (∼150 µs reduced to ∼120 µs), becomes less distinct.

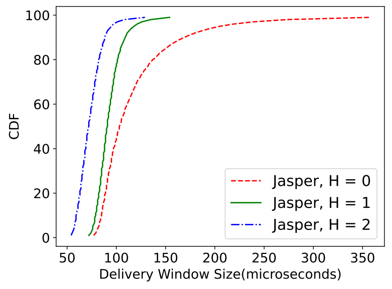
Reduced OWDs for each receiver. We further investigate the impact of hedging on the OWD of each receiver. To do so, we need to design more controlled experiments to compare the hedging case and no-hedging case. Previously we have run multiple experiments for hedging cases and no-hedging cases separately. However, we later realize that the performance variance (latency fluctuations) is significant across multiple runs, leading to inconsistent results among the multiple runs for lower percentiles. To eliminate the noise effect, we design a strategy to enable us to test two techniques (hedging and no hedging) at the same time on the same VMs. We make Jasper only do hedging for odd-numbered messages. This interleaves the hedging experiment with the no-hedging experiment, so that the OWD samples for both hedging and no-hedging cases are collected from the same experiment setting. In this way, we are able to obtain consistent and reproducible results across multiple runs. We see that hedging does not greatly reduce the median OWD (Figure 8a) experienced by each receiver which also explains the reason for the inconsistent results before we adopt interleaving. However, our hedging technique can substantially reduce the heavy tail of latency. As shown in Figure 8b, the hedging significantly reduces the 99th percentile OWD compared to no hedging on a tree.



(a) Hedging reduces overall multicast latency (OML)

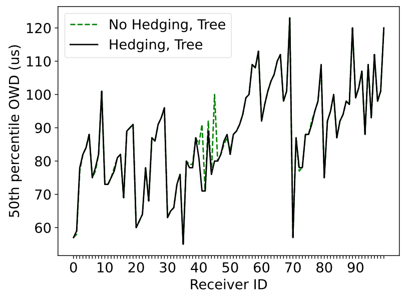


(b) Hedging shows consistently low OML

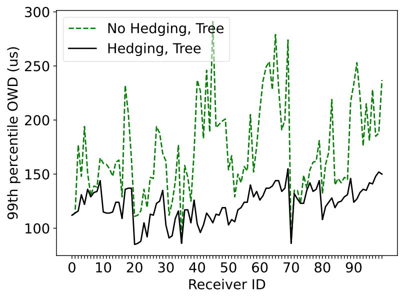


(c) Hedging reduces delivery window size

Figure 7: Evaluating VM Hedging



(a) 50th percentile OWD per receiver does not improve with hedging



(b) 99th percentile OWD per receiver improves significantly with hedging

Figure 8: Spatial OWD In Jasper

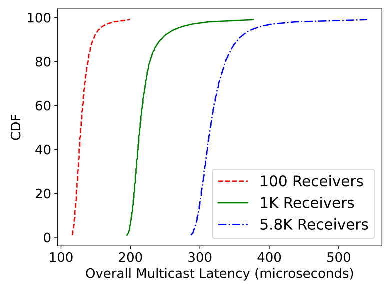
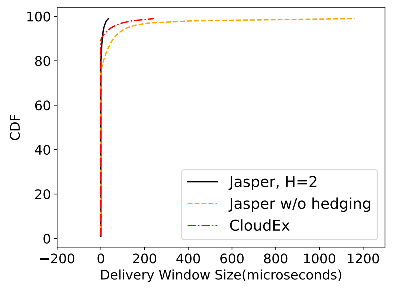
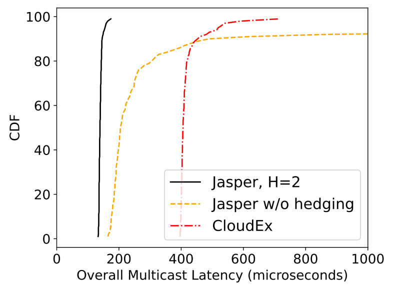


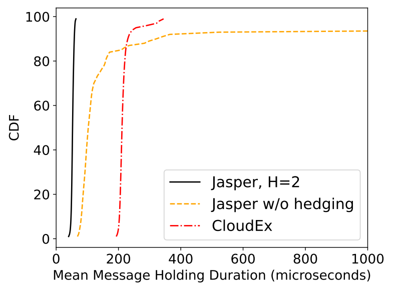
Figure 9: Jasper scales to a large number of receivers while maintaining a median OML of less than 350μs



(a) Jasper achieves a substantially narrow message delivery window size



(b) CloudEx and Jasper w/o hedging shows high OML



(c) Jasper requires a low message holding duration compared to CloudEx

Figure 10: Evaluating Multicast Fairness i.e., Jasper with Hold & Release

8.3 Simultaneous Delivery

Fairness of data delivery among the market participants (MPs) is crucial for HFTs. We adopt the same fairness definition as used by on-prem financial exchanges: every MP should receive a multicast message at nearly the same time. In other words, the delivery window size of a message is negligible. We compare with CloudEx [12] that uses a similar *hold-and-release* mechanism but does not use a tree or hedging. With the help of CloudEx authors, we faithfully implement a DPDK-based version of CloudEx that yields higher performance.

Close to ideal fairness. Figure 10a shows that Jasper can achieve a delivery window size (DWS) of 0 at very high percentiles. Without hedging, the DWS becomes larger. CloudEx also achieves fairness, but at the cost of high end-to-end latency as we show next.

Effect of fairness on latency. Figure 10b shows that the overall OWD increases significantly if no hedging is employed while using the *hold-and-release* mechanism. That is because the calculated deadlines are far into the future in order to cover the high spatial variance of latency when $H = 0$. It leads to high holding duration at each receiver before messages are released to the application. CloudEx also shows high OMLs, worse than Jasper with or without hedging, especially at ≤ 80th percentile.

Required holding duration for fairness. When a multicast receiver receives a message before the deadline, it holds the message until the deadline before processing it. Figure 10c shows the mean holding duration required by each receiver VM for achieving fair delivery. Jasper with hedging leads to a very low holding duration (~60μs) while Jasper with no hedging and CloudEx needs more holding duration which in turn leads to high multicast latency as we discussed earlier.

8.4 Fairness Impact on Financial Trading

To illustrate the impact of fairness on financial trading and further demonstrate the benefit of Jasper in preserving fairness, we build and run a prototype financial exchange application atop Jasper. We use 100 MPs as multicast receivers but only 4 receivers, i.e., the first 2 and the last 2 out of the 100 MPs, actively take part in trading. We make the MPs use the same trading algorithm and analyze their portfolios after a few rounds of trades. Further detail of this sample application is included in Appendix C. In general, the fair delivery of market data would lead to all the participants (P1-P4) to have a similar portfolio after the trading, while the unfair delivery of data would show differing portfolios (because some MPs gain an advantage over the others in the trading). Table 2 displays the results with different multicast solutions where Jasper yields almost ideal results.

	P1	P2	P3	P4
Ideal	25%	25%	25%	25%
Jasper	25%	24.97%	24.66%	25.36%
DU	49.79%	50.21%	0%	0%
Tree	56.95%	42.39%	0.40%	0.25%

Table 2: Share of overall profit earned by each of the four participants in a sample trading application. Jasper achieves closest to ideal performance.

8.5 Supporting Large Number of Receivers

In contrast to the previous multicast solutions that only work with $O(10)$ receivers (e.g., [15, 16, 38]), Jasper aims to implement a more scalable multicast service which can support $O(1000)$ receivers. Figure 9 plots Jasper's multicast latency for 100 receivers, 1K receivers, and ≈5K receivers. Because of the tree structure, Jasper can support multicast to thousands of receivers. Besides, the VM hedging enables Jasper to keep a graceful growth of latency as the number of receivers is increased by an order of magnitude.

	H=1		H=2	
N	DWS (μs)	P(F) (%)	DWS (μs)	P(F) (%)
100	0	92	0	92
5000	8	36	0	86

Table 3: Jasper achieves a low median Delivery Window Size (DWS) and a high probability of perfect fairness (P(F)). A low DWS and high P(F) are better.

Fairness for an increasing number of receivers. Table 3 shows that Jasper achieves a very low delivery window size (DWS) even with 5000 receivers. A DWS of x μs denotes that every receiver gets the message within x μs of each other. The probability of perfect fairness is also shown in the table. Perfect fairness translates to DWS being 0. The probability of fairness (P(F)) is calculated as the n-th percentile at which the DWS is 0 in a CDF. We achieve a fairness probability of 86% for 5000 receivers when using the hedging factor of 2. We note that at a high number of receivers, $H = 1$ performed poorly. We recommend using $H = 2$ for any $N \leq 5000$.

8.6 Jasper Throughput

In our current implementation of Jasper, a single multicast tree can support the rate of 35K multicast messages per second (MPS) when using $H = 2$ and $N = 100$. Beyond this, we start seeing a high packet drop rate (an increase to 0.2% from the normal packet drop rate of ≈0%). At 35K MPS with $H = 2$, each proxy sends out 1.05 million packets per second. While we are not exhausting the egress bandwidth at this rate, we suspect that the rise in packed drop rate might be because of limits enforced by the cloud provider [13]. We leave further investigation as future work. At 35K MPS, we achieve a probability for perfect fairness of 96% with 100 receivers.

8.7 Latency Impact from Different Components of Jasper

We study OML when each Jasper component is used i.e., starting from DU to using a tree, and then introducing hedging into the tree, and finally enabling simultaneous delivery

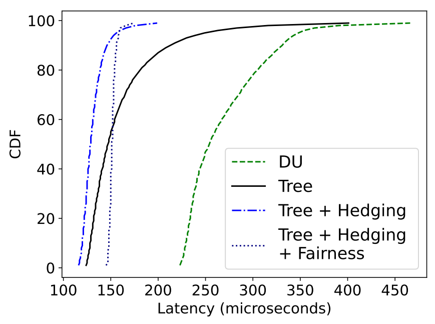


Figure 11: Impact on multicast latency as each Jasper component is introduced

for fairness. Figure 11 shows how the latency is reduced when introducing different Jasper components except for the final step of enforcing fairness. In the final step, the *hold-and-release* mechanism is enabled to preserve fairness, which increases the latency by a marginal amount to make the delivery window small.

9 RELATED WORK

Multicast: Prior works on multicast [3, 20, 27, 32, 35] mainly focus on finding optimal paths in a network (or overlay mesh) using cost models for network *links* that capture link bandwidth or latency characteristics. By contrast, Jasper focuses on minimizing the delays incurred at *hosts* while transmitting multiple message copies. In a single cloud region, triangle inequality violations of latency rarely occur, so finding the best path in a mesh of VMs isn't very helpful (§2.2).

Collective communication: Collective communication [47–49] makes use of overlay trees. However, these systems span a limited number of receivers, usually less than 100. For instance, Hoplite [49] presents a tree-based topology for bandwidth optimization and results for $O(10)$ nodes. Although Jasper also aims to achieve high throughput by judicious use of egress bandwidth, our primary goals revolve around latency (partly because message sizes are small) and its variance and involve $O(1000)$ nodes. Further, Jasper has fair delivery as a primary goal which is not a concern for either collective communication or the existing multicast literature.

Fair delivery: CloudEx [12] introduces the *hold-and-release* mechanism for fairness for financial exchanges running in the cloud. However, CloudEx fails to scale to 1000s or even to 100s of market participants because the overall latency and latency variance build up rapidly when the CES directly sends the market data to all the MPs. Octopus [15] implements a similar approach with SmartNICs to achieve fair delivery but has only been shown to scale to less than 10 receivers. Jasper scales to 1000s of emulated receivers while maintaining a low end-to-end latency. Besides, some recent literature, including DBO [16], FBA [5], and Libra [25], propose to use a modified definition of fairness in financial trading, which requires more research before such definitions may be adopted.

10 DISCUSSION AND FUTURE WORKS

Security implications of hold-and-release mechanism. To preserve fairness, Jasper's *hold-and-release* mechanism requires the receivers to hold the message until the message's deadline. Such a design may raise security concerns: the malicious receivers (MPs) may release the message earlier than the deadline to create unfair advantages to some MPs.

To tackle this problem, in Jasper we assume all the receiver VMs are owned/controlled by the financial exchange instead of MPs. MPs are only given containers to run their trading programs inside the receiver VMs with controlled network access. As a result, Jasper's security relies on the container's isolation guarantee, which has been well studied in the recent literature [2, 17, 22]. We plan to further analyze and evaluate our security assumptions in the future.

Reuse Jasper's multicast tree for order submission. A typical financial exchange includes two protocols, e.g., ITCH and OUCH [43]. ITCH corresponds to the multicast of market data from CES to MPs, whereas OUCH is related to the trade order submissions from MPs to CES. Jasper's multicast service helps to implement high-performance ITCH protocol, and we have not considered optimizing OUCH with Jasper in this paper. We believe Jasper's multicast tree structure can also be leveraged to achieve high throughput for the OUCH protocol: When a large number of MPs are submitting orders concurrently, the incast traffic can lead to the bottleneck at the CES. To solve this, the financial exchange can employ the proxies in Jasper to aggregate the incoming orders in a hierarchical way, thus alleviating the burden of CES. Besides, we expect Jasper can also reduce the latency variance in the OUCH direction so as to improve the fairness among MPs during order submission (i.e., the inbound fairness defined by CloudEx [12]). We leave it as our future work to study using Jasper to improve OUCH protocol of the financial exchange.

11 CONCLUSION

The migration of financial exchanges to the public cloud imposes several networking challenges. This paper presents Jasper, a cloud-native multicast service to work for financial exchanges in the cloud environment. Jasper can achieve low latency and fair data delivery when multicasting to a large number (1000s) of emulated receivers. The performance of Jasper comes from three main components: (i) a proxy tree to minimize replication and transmission delay of multiple copies of the multicast message, (ii) a VM hedging technique to reduce the latency variance and narrow down the window in which each MP receives the message copy of a multicast message and, (iii) a message *hold-and-release* mechanism where deadlines are attached to messages and receivers only processes the messages at or after the deadlines for fairness.

This work does not raise any ethical issues.

REFERENCES

[1] [n. d.]. Matching Orders Definition. https://www.investopedia.com/terms/m/matchingorders.asp. Accessed: 2021-02-02.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[3] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. 2002. Scalable application layer multicast. *SIGCOMM Comput. Commun. Rev.* 32, 4 (aug 2002), 205–217. https://doi.org/10.1145/964725.633045

[4] S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller. 2003. Construction of an efficient overlay multicast infrastructure for real-time applications. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, Vol. 2. 1521–1531 vol.2. https://doi.org/10.1109/INFCOM.2003.1208987

[5] Eric Budish, Peter Cramton, and John Shim. 2015. The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response *. *The Quarterly Journal of Economics* 130, 4 (07 2015), 1547–1621. https://doi.org/10.1093/qje/qjv027 arXiv:https://academic.oup.com/qje/article-pdf/130/4/1547/30637414/qjv027.pdf

[6] Sara Castellanos. [n. d.]. Nasdaq Ramps Up Cloud Move. https://www.wsj.com/articles/nasdaq-ramps-up-cloud-move-11600206624. Accessed: 2024-01-31.

[7] Tatsuhiro Chiba, Toshio Endo, and Satoshi Matsuoka. 2007. High-Performance MPI Broadcast Algorithm for Grid Environments Utilizing Multi-lane NICs. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 487–494. https://doi.org/10.1109/CCGRID.2007.59

[8] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[9] Arijit Ganguly, P. Oscar Boykin, and Renato Figueiredo. 2010. Techniques for low-latency proxy selection in wide-area P2P networks. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8. https://doi.org/10.1109/IPDPSW.2010.5470939

[10] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2022. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. *Proc. VLDB Endow.* 16, 4 (dec 2022), 629–642. https://doi.org/10.14778/3574245.3574250

[11] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, Berkeley, CA, USA, 81–94.

[12] Ahmad Ghalayini, Jinkun Geng, Vighnesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. 2021. CloudEx: A Fair-Access Financial Exchange in the Cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 96–103. https://doi.org/10.1145/3458336.3465278

[13] GitHub. [n. d.]. Performance in terms of PPS. https://github.com/amzn/amzn-drivers/issues/68.

[14] Piotr J. Gmytrasiewicz and Edmund H. Durfee. 1992. Decision-theoretic recursive modeling and the coordinated attack problem. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems* (College Park, Maryland, USA). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 88–95.

[15] Junzhi Gong, Yuliang Li, Devdeep Ray, KK Yap, and Nandita Dukkipati. 2024. Octopus: A Fair Packet Delivery Service. *arXiv preprint arXiv:2401.08126* (2024).

[16] Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. 2023. DBO: Fairness for Cloud-Hosted Financial Exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 550–563. https://doi.org/10.1145/3603269.3604871

[17] The gVisor Authors. [n. d.]. gVisor: The Container Security Platform. https://gvisor.dev/. Accessed: 2023-01-18.

[18] Michael T. Helmick and Fred S. Annexstein. 2007. Depth-Latency Tradeoffs in Multicast Tree Algorithms. In *21st International Conference on Advanced Information Networking and Applications (AINA '07)*. 555–564. https://doi.org/10.1109/AINA.2007.52

[19] Owen Hilyard, Bocheng Cui, Marielle Webster, Abishek Bangalore Muralikrishna, and Aleksey Charapko. 2023. Cloudy Forecast: How Predictable is Communication Latency in the Cloud? *arXiv preprint arXiv:2309.13169* (2023).

[20] Yang hua Chu, S.G. Rao, S. Seshan, and Hui Zhang. 2002. A case for end system multicast. *IEEE Journal on Selected Areas in Communications* 20, 8 (2002), 1456–1471. https://doi.org/10.1109/JSAC.2002.803066

[21] Intel. [n. d.]. DPDK Programmer's Guide: Ring Library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html. Accessed: 2024-01-31.

[22] Intel James E Chamings. [n. d.]. Intel Clear Containers. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-clear-containers-1-the-container-landscape.html. Accessed: 2023-01-18.

[23] Cristian Lumezanu, Randy Baden, Neil Spring, and Bobby Bhattacharjee. 2009. Triangle inequality variations in the internet. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (Chicago, Illinois, USA) *(IMC '09)*. Association for Computing Machinery, New York, NY, USA, 177–183. https://doi.org/10.1145/1644893.1644914

[24] Donald MacKenzie. 2021. *Trading at the speed of light: How ultrafast algorithms are transforming financial markets.* Princeton University Press.

[25] Vasilios Mavroudis and Hayden Melton. 2019. Libra: Fair Order-Matching for Electronic Financial Exchanges. arXiv:1910.00321 [cs.CR]

[26] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 537–550. https://doi.org/10.1145/2785956.2787510

[27] Kianoosh Mokhtarian and Hans-Arno Jacobsen. 2015. Minimum-Delay Multicast Algorithms for Mesh Overlays. *IEEE/ACM Transactions on Networking* 23, 3 (2015), 973–986. https://doi.org/10.1109/TNET.2014.2310735

[28] NASDAQ. [n. d.]. Nasdaq TotalView-ITCH 5.0. https://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTVITCHSpecification.pdf. Accessed: 2024-02-02.

[29] Nasdaq.com. [n. d.]. Nasdaq and AWS Partner to Transform Capital Markets. https://www.nasdaq.com/press-release/nasdaq-and-aws-

partner-to-transform-capital-markets-2021-12-01. Accessed: 2024-01-26.

[30] Brian Nigito. [n. d.]. Multicast and the Markets. https://signalsandthreads.com/multicast-and-the-markets. Accessed: 2024-01-30.

[31] Alexander Osipovich. [n. d.]. Google Invests 1 Billion in Exchange Giant CME, Strikes Cloud Deal. https://www.wsj.com/articles/google-invests-1-billion-in-exchange-giant-cme-strikes-cloud-deal-11636029900. Accessed: 2021-02-02.

[32] M. Parsa, Qing Zhu, and J.J. Garcia-Luna-Aceves. 1998. An iterative algorithm for delay-constrained minimum-cost multicasting. *IEEE/ACM Transactions on Networking* 6, 4 (1998), 461–474. https://doi.org/10.1109/90.720901

[33] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2021. When to Hedge in Interactive Services. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 373–387. https://www.usenix.org/conference/nsdi21/presentation/primorac

[34] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2021. When to Hedge in Interactive Services. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 373–387. https://www.usenix.org/conference/nsdi21/presentation/primorac

[35] G.N. Rouskas and I. Baldine. 1997. Multicast routing with end-to-end delay and delay variation constraints. *IEEE Journal on Selected Areas in Communications* 15, 3 (1997), 346–356. https://doi.org/10.1109/49.564133

[36] Vighnesh Sachidananda. [n. d.]. LemonDrop. https://stacks.stanford.edu/file/druid:xq718qd4043/Vig_thesis_submission-augmented.pdf. Accessed: 2024-01-30.

[37] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. 2009. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.* 35, 12 (2009), 581–594. https://doi.org/10.1016/j.parco.2009.09.001 Selected papers from the 14th European PVM/MPI Users Group Meeting.

[38] Amazon Web Services. [n. d.]. Multicast on transit gateways. https://docs.aws.amazon.com/vpc/latest/tgw/tgw-multicast-overview.html.

[39] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. 2019. Elmo: source routed multicast for public clouds. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 458–471. https://doi.org/10.1145/3341302.3342066

[40] Andrew Smith. 2014. Fast money: the battle against the high frequency traders. *The Guardian* 7 (2014).

[41] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[42] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low latency via redundancy. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (Santa Barbara, California, USA) *(CoNEXT '13)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2535372.2535392

[43] Jianling Wang, Vivek George, Tucker Balch, and Maria Hybinette. 2017. Stockyard: a discrete event-based stock market exchange simulator. In *Proceedings of the 2017 Winter Simulation Conference* (Las Vegas, Nevada) *(WSC '17)*. IEEE Press, Article 89, 11 pages.

[44] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 459–471. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein

[45] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 329–341. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_yunjing

[46] Jianjun Zhang, Ling Liu, Lakshmish Ramaswamy, and Calton Pu. 2008. PeerCast: Churn-resilient end system multicast on heterogeneous overlay networks. *Journal of Network and Computer Applications* 31, 4 (2008), 821–850. https://doi.org/10.1016/j.jnca.2007.05.001

[47] Liangyu Zhao and Arvind Krishnamurthy. 2023. Bandwidth Optimal Pipeline Schedule for Collective Communication. arXiv:2305.18461 [cs.NI]

[48] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2023. Efficient Direct-Connect Topologies for Collective Communications. arXiv:2202.03356 [cs.NI]

[49] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 641–656. https://doi.org/10.1145/3452296.3472897

A APPENDIX: DE-DUPLICATION IMPLEMENTATION WITH SOCKET+EBPF

As described in §7.2, we implement the socket API in Jasper for usability and use eBPF to reduce the overheads of message duplication detection at the receivers. Here we describe the details of our implementation.

We provide a mechanism to implement a set membership check using a fixed-length buffer in eBPF/XDP. The main assumption that enables this is that packet re-ordering should not interleave packets that are sent a second apart. A similar assumption has been made in [44] albeit in a different context where packets sent 10 milliseconds apart are assumed to never get re-ordered.

We start with a fixed-length buffer B of size N where N equals the smallest power of 2 that is greater than or equal to the message rate, R (i.e., R messages per second). This restriction on N allows us to implement modulo operation, not allowed in XDP, as a bitwise operation. The buffer is implemented as a bpf_map of size N. Algorithm 1 shows how we achieve packet de-duplication and only accept the packet copy that is received earliest and discard the rest.

Algorithm 1 Packet De-Duplication with a Fixed-Length Buffer

```
1:  Input: Message ID ($id$), Buffer $B$ of size $N$
2:  procedure PacketDeduplication($id, B, N$)
3:      $index \leftarrow id \ \& \ (N-1)$
4:      if $B[index] < id$ then
5:          $B[index] \leftarrow id$
6:          Accept the message
7:      else if $B[index] == id$ then
8:          Discard the message as a duplicate
9:      else
10:         Error: $B[index] > id$        ▷ Assumption Violated
11:     end if
12: end procedure
```

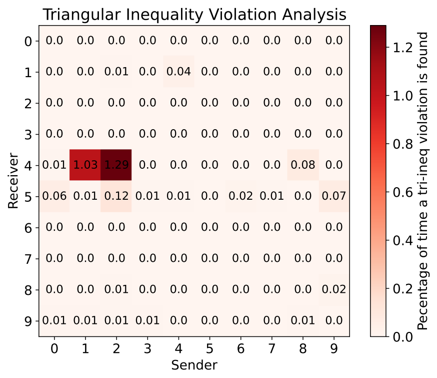


Figure 12: Triangular inequality violations are rare in GCP.

B APPENDIX: LATENCY CHARACTERIZATION IN GCP

Figure 12 shows that triangular inequality violations for path latencies are rare in GCP.

C APPENDIX: FINANCIAL EXCHANGE SETUP FOR SIMULTANEOUS DELIVERY DEMONSTRATION

(1) Below we describe the trading algorithm run by each MP in §8.4. There is one sender and 100 receivers involved in this trading setup.
(2) The sender creates trades, and each trade contains information such as the type of trade (**BUY** or **SELL**), the trade symbol, the number of shares, and the price per share.
(3) The first and last two receivers (market participants) are the ones actively participating in the trading.
(4) All participants use a similar trading algorithm: If the price per share goes above a specific threshold, each of the active participants places an order for the corresponding trade symbol. They also include the current timestamp when submitting their orders.
(5) The CES receives the orders from the MPs and processes the orders using continuous matching algorithm [1]. When the sender receives multiple trades for the same trade symbol, only the order with the earliest timestamp is processed.
(6) After processing 5000 orders, the profit earned by each market participant is evaluated and shown in Table 2