

# Network Support For Scalable And High-Performance Cloud Exchanges

Muhammad Haseeb  
New York University, USA

Jinkun Geng  
Stanford University, USA

Daniel Duclos-Cavalcanti  
Technical University of Munich,  
Germany

Xiyu Hao  
New York University, USA

Ulysses Butler  
New York University, USA

Radhika Mittal  
University of Illinois  
Urbana-Champaign, USA

Srinivas Narayana  
Rutgers University, USA

Anirudh Sivaraman  
New York University, USA

## ABSTRACT

Financial exchanges are migrating to the public cloud, but the best-effort nature of the cloud fabric is at odds with the stringent networking requirements of the exchanges. We present Onyx, a system for meeting such requirements which uses many well-studied techniques in a new context as well as introduces new techniques that enable a scalable cloud financial exchange. An overlay multicast tree is used to disseminate data to 1000 participants with  $\leq 1 \mu\text{s}$  difference in data reception time between any two participants, crucial for maintaining fair competition. Several techniques for mitigating latency variance are introduced. Onyx also presents a scheduling policy for trade orders that enhances an exchange's performance and gracefully services bursty traffic. Onyx achieves  $\approx 50\%$  lower latency than the AWS multicast service [1]. Onyx outperforms an existing system, CloudEx [2] in terms of supported number of participants, exchange's throughput and multicast latency. Onyx's techniques can be applied to other existing systems (e.g., DBO) to enhance their performance.

## CCS CONCEPTS

• **Networks** → **Network algorithms; Network protocol design; Application layer protocols; Logical nodes; Packet scheduling; Cloud computing; Overlay and other logical network structures;**

## KEYWORDS

Cloud Financial Exchanges, Overlay Multicast, Low Latency Architecture, Packet Scheduling, Scalable Exchange

### ACM Reference Format:

Muhammad Haseeb, Jinkun Geng, Daniel Duclos-Cavalcanti, Xiyu Hao, Ulysses Butler, Radhika Mittal, Srinivas Narayana, and Anirudh Sivaraman. 2025. Network Support For Scalable And High-Performance Cloud Exchanges. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25)*, September 8–11, 2025, Coimbra, Portugal. ACM, New York, NY, USA, 22 pages. <https://doi.org/10.1145/3718958.3750530>

## 1 INTRODUCTION

Financial exchanges are migrating to the public cloud for reasons such as improved scalability and reduced capital expenditure. Despite its benefits, the public cloud poses unique challenges. Exchanges have traditionally operated in on-premise or colocation facilities, engineered for deterministic and low latency. For fair market access, exchanges equalize cables, while employing low-jitter switches [3], between the exchange and participant servers. This approach ensures that (i) all participants receive market data from the exchange simultaneously (*outbound fairness*) and (ii) an order generated earlier by one participant reaches the exchange before orders generated later by other participants (*inbound fairness*).

However, the public cloud lacks these enhancements. It is a best-effort environment characterized by nondeterminism (e.g. latency variance in Figure 1). In response, several projects have developed techniques for cloud-based exchanges. These include using synchronized clocks to compensate for nondeterminism [2], using SmartNICs to hold data until all receivers have received it [4], and new fairness definitions [5, 6]. These projects have demonstrated promising results for tens of participants but exhibit significant performance degradation as the number of participants increases. This limitation arises because scalability was not a primary design objective. Instead, the initial focus was on establishing a functional proof of concept for a fair exchange on the cloud. Having made significant progress in that regard, the next logical step is to consider scalability. Consequently, our paper develops techniques for *scale: How do we design a network to support communication between the exchange and ~1000 participants in the cloud while ensuring fairness in the exchange?*<sup>1</sup> Further, we explore the answer to the above question from the perspective of a cloud tenant, i.e., whether the customers of public cloud can build a scalable financial exchange atop the cloud *without* requiring special help or hardware access from the cloud provider. In practice, a scalable exchange would also need scale-out *compute* techniques for the exchange server which we place out of scope for this paper.

Our system, Onyx, tackles two major challenges to provide network support for scalable exchanges. First, how do we support *outbound* communication of market data (information about the

<sup>1</sup>We target 1000 as a number that is sufficiently larger than the typical number of participants ( $\approx 100$ ) supported by on-premises exchanges [7, 8].

state of market) from an exchange to 1000 market participants, while ensuring (a) low spatial variance, i.e., all participants receive market data nearly simultaneously, (b) low latency from the exchange server to the participant, and (c) low temporal variance, i.e., latency doesn't fluctuate over time? Second, how do we support *inbound* communication of participant orders to the exchange while (a) providing chances of trades to all the traders fairly and, (b) achieving high throughput for the exchange, especially during intense market activity when bursts of orders arrive, causing incast-style [9] drops? Onyx integrates well-established mechanisms within a new use case as well as introduces novel techniques to design scalable exchanges.

First, to scale outbound communication from the exchange to a large number of participants, we employ an overlay multicast tree composed of a *root* exchange VM, proxy VMs as *intermediate* nodes, and participant VMs as *leaf* nodes. We develop a simple and effective heuristic to tune the tree's fan-out and depth, navigating a trade-off between increased serialization delay due to greater fan-out and increased propagation delay because of greater depth. Higher fan-out leads to higher serialization/transmission delay as outgoing messages gets serialized by the NIC so earlier messages have noticeable lower latency than the later ones. Higher depth of the tree introduces more VMs in the path of messages which increases the overall latency. To lower latency and counter the cloud's variability, we pervasively employ *hedging*: routing redundant copies of the market data through multiple proxy and receiver VMs and rotating parent-to-child associations at each tree level on each multicasted packet.

Second, to provide scalable and fair inbound communication from participants to the exchange, we propose a sequencer that relies on recent advancement in clock synchronization [10] that is robust to latency fluctuations and works for VMs without hardware support. The sequencer ensures that an exchange server sees the messages generated by participants in their generation order so that *inbound fairness* is achieved regardless of the arbitrary message delays from participants to the exchange. We also propose a scheduling policy, Limit Order Queue (LOQ), that helps achieve fairness and high order matching rate of the exchange during bursty market activity. Finally, we reuse the overlay multicast tree in the *reverse direction* to relay participants messages to the exchange which helps against incast-style packet drops by reducing the fan-in at the exchange when supporting a large number of participants. This leads to high throughput for the exchange server and low latency for trade orders.

Onyx can support a maximum throughput of 175K multicast messages per second, at which point it is limited by a proxy VM's egress bandwidth. It also scales to 1000 receivers/VMs, achieving a median multicast latency of  $\leq 250 \mu\text{s}$  while maintaining a latency difference of  $\leq 1 \mu\text{s}$  across these receivers. Onyx supports more participants (1K as opposed to 0.1K) and achieves 50% lower latency compared to AWS' Transit Gateway-based multicast. On the inbound side, Onyx efficiently handles large bursts of orders maintaining low latencies. Onyx outperforms a prior system, CloudEx [2] in terms of scalability, order matching rate and multicast latency. More importantly, the techniques described in Onyx are meant to provide a networking layer for the exchanges and are thus composable with the existing systems [2, 6] to enhance their performance.

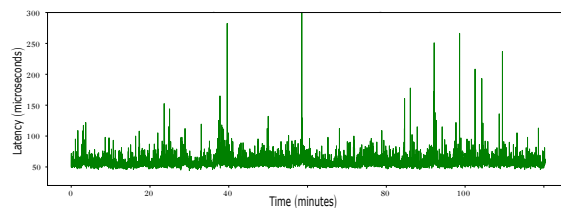


Fig. 1: Latency between a pair of VMs varies over time

To contextualize Onyx's absolute performance, it falls short of heavily engineered on-premises financial exchanges, which achieve latency differences of tens of nanoseconds across receivers using low-jitter switches and equalized cables connecting colocated participants. However, participant-to-exchange interfaces exist along a *performance-usability tradeoff curve*: colocated participants lie at one end, paying high premiums for direct exchange connectivity and limited in number, while web market data APIs [11] lie at the other end, offering low cost and ease of access at the expense of performance with no fairness guarantees. Onyx offers a compelling point on this curve: it delivers low and predictable latencies, with  $\leq 1 \mu\text{s}$  latency differences across receivers, while scaling to 1K participants and offering many additional benefits of the public cloud. Appendix L further discusses Onyx's position relative to on-premises systems. Onyx will be open-sourced.

## 2 BACKGROUND

**Financial Exchange Setup.** An exchange typically has an exchange server and multiple market participant (*MP*) servers. The exchange server runs a matching engine (*ME*) to process trading orders from the MPs and multicast market data to them. Market data reveals market state e.g., asset prices and processed orders. Orders can be bid orders, which aim to purchase an asset at a specific price, and ask orders, which aim to sell an asset at a specific price.

The ME maintains a limit order book (*LOB*) (Fig. 4), which lists all bid and ask orders from MPs. When a bid order's price exceeds or matches an ask order's price, the two orders are executed (and matched together) using some matching algorithm. Unexecuted orders remain in the LOB, waiting for a match. In the price-time-priority matching algorithm [12],<sup>2</sup> orders are arranged in price levels, with those at the same level sorted by their arrival time at the exchange (which is equivalent to sorting them by their generation time if all cables connecting MPs to exchange are equal). An LOB snapshot (Fig. 4) shows a separation between bid and ask price levels, with the mid-price indicating an asset's true value. Orders closer to the mid-price have higher chances of getting matched early. Any scheduling policy (e.g., LOQ) on orders should ensure that the semantics of matching algorithm are not impacted i.e., the sequence of matched orders should remain unchanged.

**Challenges of Cloud Migration.** While the public cloud offers many advantages, it does not offer low-level control, e.g., allowing tenants to control wire lengths or employing low jitter switches. The public cloud also exhibits high latency variance [14]. The latency between

<sup>2</sup>It is a widespread matching algorithm, so we use it. Alternatives exist [13].

one pair of VMs can be significantly different from another pair [15]. Latency also fluctuates over time: figure 1 plots the 90p latency between a pair of VMs in an AWS region for a tumbling window of 1 s. The figure also shows infrequent but unpredictable latency spikes that substantially increase latency [6, 14]. Such phenomena make it challenging to achieve low and deterministic latency in the cloud.

Given the above challenges, it is reasonable to ask whether financial exchanges should ever be migrated to the cloud – and if so, how. This is an ongoing debate with reasonable arguments on both sides. Beyond the research projects [2, 4, 6], some cloud providers are exploring close partnerships to build private clouds for exchanges’ bespoke requirements [16, 17]. Onyx contributes to this debate by studying what performance guarantees can be achieved if a *cloud tenant designs the exchange architecture on the public cloud with publicly available cloud APIs*. With Onyx’s DIY approach, a tenant does not need to wait for a cloud provider to build private clusters.

**Is cloud migration still relevant?** Debates surrounding cloud migration have also emerged, particularly regarding its effectiveness as a panacea for cost reduction. This discourse has contributed to the rise of *cloud repatriation* –the practice of moving workloads back from the cloud to on-premises [18]. Cloud repatriation, unlike cloud migration, is highly relevant to large SaaS companies like Snowflake or Dropbox, which operate at a scale where they can benefit from economies of scale by owning infrastructure and have consistent workloads that justify such investments. However, this narrative doesn’t extend as well to smaller entities like financial institutions, which lack the consistent, cloud-scale workloads to make private infrastructure cost-effective and often benefit more from the flexibility and cost savings of not managing their own hardware.

**Prior Work and Onyx’s Motivation.** CloudEx [2] is among the earliest systems designed for cloud-based exchanges and capitalizes on accurate clock synchronization. To maintain outbound fairness, MPs wait until a set timeout before processing any order to ensure every MP has received the market data. For inbound fairness, the ME waits until a set timeout before processing any order to ensure that all the earlier generated orders have been received and ordered by their generation timestamps. The timeouts if too big, lead to low performance and if too small, run the risk of violating fairness. As the number of participants increase, tuning the timeouts become difficult as avenues for latency variance and straggler behavior increases.

DBO [6] leverages mechanisms to always guarantee fairness among participants by decoupling fairness from latency fluctuations, although it is only applicable to a subset of trades that depend on the last received market data batch. As the number of participants increase, DBO also suffer from degraded performance because of (i) incast on the inbound side and, (ii) large latency of market data because of increased transmission delay. Both CloudEx and DBO can benefit from our techniques to achieve high performance at scale.

Onyx’s main contribution is architecting for scale while maintaining fairness and achieving high performance. Onyx adopts the idea of synchronized clocks from CloudEx, but scales the system much further by using a communication tree in both the inbound

and outbound directions to achieve high performance with a large number of participants. Onyx augments the tree with (i) a message sequencing mechanism to ensure inbound fairness under latency fluctuations (ii) a scheduling algorithm to achieve high performance under bursts of orders and, (iii) several variance mitigation techniques to achieve fairness.

### 3 ONYX OVERVIEW

**Setup and goals.** Each MP VM ( $VM_i$  in fig. 2) hosts an order gateway and a trading algorithm.<sup>3</sup> Clocks of all MP VMs and the exchange server are synchronized using Huygens algorithm [10], enabling nanoseconds level synchronization for VMs without hardware support. A trading algorithm generates the orders and submits it to the colocated gateway (in the same VM). A gateway attaches its current timestamp to an order and forwards it to the exchange (via TCP [19]), which hosts the matching engine (ME) to process orders. MP VMs are controlled by the exchange which loads the trading algorithms in them. Market data from the exchange (via UDP [19]) first arrives at a gateway which then forwards it to the trading algorithm. Such a model has been proposed previously [8, 20].<sup>4</sup> Onyx aims to provide fairness during the competition among MPs for both the outbound (exchange to MPs) and inbound (MPs to exchange) directions.

*Definition 3.1 (Outbound Fairness).* Every market data message sent from the exchange server to the market participants (MPs) should be seen by all the MPs simultaneously.

*Definition 3.2 (Inbound Fairness).* An order generated earlier than other orders should be processed by the matching engine earlier than the other orders, irrespective of which MP generated which order.

Given the above definitions, what existing systems and Onyx achieve is an approximation: (i) for outbound fairness, the difference between multicast latency between any two participants is reduced as much as possible and (ii) for inbound fairness, orders should be executed in the order of their generation-timestamps (while having synchronized clocks). Onyx further aims to achieve high performance i.e., supporting a larger number of MPs and achieving a higher exchange throughput than the previous works while achieving fairness. Figure 2 presents Onyx’s architecture. Onyx employs a bidirectional overlay tree: the exchange server is the root and MPs are the leaves, with intermediate proxy nodes. It is well known that trees help scale communication to many receivers [21, 22]. We build on such a tree to provide an *overlay multicast service* for market data and handle order submissions from a large number of MPs, but adapt the tree to the high-variance environment of the public cloud.

**Outbound: ME to Participants (market data multicast).** On the outbound side of an exchange, we augment the base tree with 3 techniques to lower multicast latency and minimize the latency variance: (i) round-robin packet spraying, (ii) proxy hedging and (iii) receiver hedging. All 3 hedge the risk of some part of the system exhibiting performance variance so that the overall multicast

<sup>3</sup>“trading algorithm” and “MP” are interchangeably used

<sup>4</sup>A brief discussion of several deployment/trust models and their performance is presented in Appendix G.

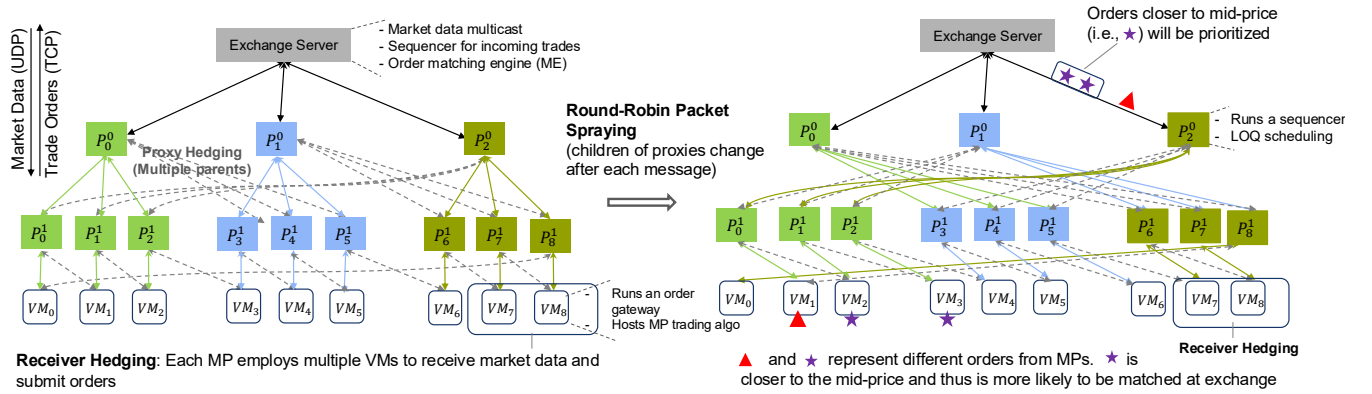


Fig. 2: Overview of Onyx.

latency as well as latency difference across receivers is sufficiently reduced.

*Round-robin packet spraying* helps with reducing impact of latency spikes on the links as paths of messages are continuously changed. *Proxy hedging* mitigates the impact of straggler proxy nodes as each child node receives duplicate messages from aunt nodes. Lastly, *receiver hedging* hedges against the risk of a receiver VM becoming slow to process incoming messages by assigning two VMs to each trader/participant running the same trading algorithm.

**Inbound: Participants to ME (orders submission).** We develop a sequencer that sits at the ingress of the exchange. All the incoming orders from MPs are fed to the sequencer while the output (the sequenced orders) is processed by the matching engine (ME) hosted by the exchange server. The sequencer ensures the exchange sees orders in the order of their generation timestamps.

During bursty market activity, MPs generate large number of orders overwhelming the exchange. As we employ TCP on the inbound, the overwhelming of the exchange leads to queue build ups at the MP VMs as packet drops increases. A special priority queue, Limit Order Queue (LOQ), runs at the egress of each MP VM for servicing the queues so that latency of orders remains low and high order matching rate at ME is achieved, while ensuring inbound fairness. LOQ schedules the queued orders in a way that if an order is going to be executed *after* some other orders by the exchange, then it can afford to *wait longer* in the queue without affecting inbound fairness while giving a chance to other, more critical, orders to be serviced.

Further, as the number of MPs increases, the ingress of the exchange server becomes a bottleneck because of the large number of MPs submitting orders. Even if the cumulative load of all the MPs is below the ingress capacity of the exchange, the instantaneous overload leads to incast-style packet drops. We reuse the multicast tree but in the reverse direction (for MP-to-exchange communication): MPs submit their orders to their parent proxies where it travels up the tree and reaches the ME at the root. It reduces the fan-in factor of ME: the ME has to receive and process streams of orders from a small number of proxies instead of all MPs. This reduces packet losses (and resulting TCP retransmissions), and increases the exchange’s throughput. As queues may form in the tree nodes, LOQ is

used in these nodes as well. TCP connections are terminated at each proxy, allowing orders’ reordering by LOQ. For audience’s ease, all the system’s assumptions, mentioned appropriately in respective sections, are also summarized in Appendix O.

## 4 MARKET DATA MULTICAST

Due to the lack of switch support, the multicast in the cloud is typically implemented by using multiple direct unicasts. Since the back-to-back unicasts are serialized over the sender’s egress, the latter receivers (among a large number of receivers) will receive the message much later than the others due to the cumulative serialization delay at the sender. To reduce this effect, we use an overlay tree, helping us to scale number of receivers. As illustrated in Figure 2, the sender sits at the root of the tree and only sends its messages to a limited number of proxies. Each proxy then relays the message down the tree to the lower-layer proxies, and recursively down to receiver VMs. Since each node’s fanout is limited, the serialization delay is constrained at each layer, reducing the variance of message delays among receivers and reducing the multicast latency, i.e., the worst one-way delay (OWD) to any receiver as shown in Fig. 3.

Tuning the tree’s fanout ( $F$ ) and depth ( $D$ ) is crucial for minimizing multicast latency for a given number of receivers ( $N$ ) as extra hops in the path of messages increase latency. We conduct experiments with various  $\langle D, F \rangle$  for  $N = 10, 100$ , and 1000 and observe that  $F \approx 10$  and  $D = \lceil \log_{10} N \rceil$  yield sufficiently low latency, and more sophisticated strategies bring little gains beyond that (Appendix A) because of the cloud’s inherent variability.

For ensuring outbound fairness, we follow CloudEx’s “hold-and-release” technique [2] that achieves simultaneous delivery (i.e., negligible multicast latency difference across any two receivers). Here we summarize the technique:

**Hold-and-release:** Exchange attaches a deadline to all outgoing market data messages. Each order gateway (inside receiver VMs) holds the received messages and releases them to the MP’s trading algorithms at the deadline (or after it, if a message does not arrive at the gateway before the deadline). The exchange calculates the deadline for a message by adding a *headroom* to the sending time. The headroom is decided by the maximum latency from the exchange to every receiver. The exchange and the gateways use periodic probes

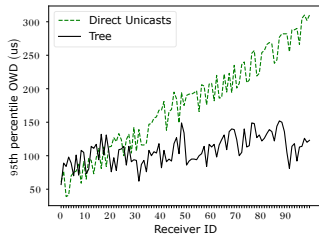


Fig. 3: Tree scales well.

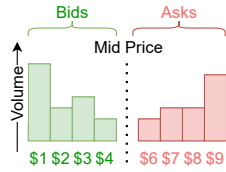


Fig. 4: LOB snapshot.

to estimate latency from the exchange to every receiver VM. Hold-and-release ensures, for most data as shown by evaluation, that each multicast message is processed by all MPs at the same time. A scalable implementation is discussed in Appendix B which uses the multicast tree in reverse to send back estimated latencies from a large number of gateways to the exchange for aiding calculation of deadlines.

Since Onyx targets a much larger number (~1000) of receivers than CloudEx (~10), simply applying the “hold-and-release” is not sufficient. As the number of receivers increases and the latency exhibits high variance, the required holding duration by a gateway increases which inflates the multicast latency. To improve scalability, we use a tree and incorporate three optimizations into our tree to reduce both the end-to-end latency and the latency variance, leading to low holding durations. As the temporal latency variance decreases, the deadline of hold-and-release become more effective.

### 4.1 Round-Robin Packet Spraying

In order to reduce latency spikes and alleviate the impact of bursts of market data, we develop round-robin packet spraying (RRPS). In RRPS, parent-child links in the tree are flexible, i.e., proxies change their children after each multicast message. Specifically, for a given proxy, on every new multicast message, the set of children is circularly shifted by 1. For example, the first proxy in a layer will have the first  $F$  proxies of the next layer as its children for message 0, the next  $F$  proxies as children for message 1, etc. This ensures that each proxy has a parent for each message, while the set of children for a parent proxy continuously changes. With RRPS, a single leaf node in a proxy tree has a total of  $\prod_{d=1}^{D-1} F^d$  different paths starting from the root node. Successive messages traverse this abundance of paths in a round-robin fashion. By contrast, without RRPS, there is only one path from the root to each leaf node and all messages traverse that path. The increased path diversity provides two major benefits for latency reduction as discussed in the following.

First, if a spike occurs on any VM-to-VM link, only a subset of messages is impacted: Since messages are round-robin across all available paths, many messages will take alternative paths that may avoid the affected link.

Second, RRPS uncovers and utilizes new paths that were unused earlier. For example, consider two adjacent proxy layers: a parent layer with  $P$  proxies and a children layer with  $P \times F$  proxies where each parent proxy has  $F$  children. Without RRPS, the number of paths utilized by messages going from parents to children is  $P \times F$ . With RRPS, there are  $P \times P \times F$  paths. This allows us to distribute

bursts among many more paths, which reduces queue build-up and latency.

RRPS works because there is an abundance of paths within an overlay proxy tree. This is because every VM in a cloud region can communicate with every other VM –effectively the network topology is a clique. This is unlike the links in physical networks’ multicast [23] where the network topology is not a clique and is limited by physical constraints like geographic distance and cables.

In Appendix K, we present a Monte-Carlo analysis exhibiting latency reduction because of RRPS when some links undergo latency fluctuations. We assume non-IID links’ latencies as it may closely reflect the conditions where only a subset of links may go through spikes at a time.

### 4.2 Proxy Hedging

A VM’s performance in the cloud can degrade due to factors like noisy neighbors or live VM migration, causing latency fluctuations for all messages passing through the VM [24]. To mitigate this, we develop proxy hedging, where each node in the tree receives multiple copies of a message from proxies in the higher layer. A node processes only the first copy and discards duplicates, thus reducing the impact of any straggler proxy VMs. It decreases both temporal and spatial latency variance, leading to more stable latency and reducing the delivery window size (i.e., the difference between the earliest and latest receiver’s latency).

Each proxy sends messages to the children of  $H$  of its siblings along with sending messages to children of its own where  $H$  is the hedging factor. For example, in Figure 2 (left), when hedging is not enabled (i.e.,  $H = 0$ ), proxy  $P_3^1$  (3<sup>rd</sup> proxy in 1<sup>st</sup> layer) 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 one (if  $H = 1$ ) other node,  $P_0^0$ . A proxy processes the earliest copy among  $H + 1$  copies of a message and discards the rest, achieving significant latency reduction.

Unlike the previous technique, proxy hedging leads to redundant work which decreases the goodput of a proxy. The goodput of a proxy node can be defined as:  $\frac{1}{H+1} \times \text{throughput}$  because each proxy sends redundant messages to the children of  $H$  siblings. One way to recover the lost throughput is to employ several parallel trees that share the root and the leaves (assuming the leaves have enough ingress bandwidth). The increased number of proxies also raises concerns of dollar cost, which could potentially be a drawback here. Appendix H includes an estimation of monetary cost. Nevertheless, latency and latency variance minimization is the primary objective in financial exchanges as it affects fairness while the throughput (on the outbound side), a less scarce resource for exchanges, can be enhanced by various orthogonal approaches.

**Choosing  $H$ :** Increasing  $H$  brings benefits i.e., latency and latency variance reduction but it also reduces the multicast message rate that can be supported because of the redundant work performed by proxies. So it is beneficial to choose a *low*  $H$  that provides sufficient performance benefits.

We empirically chose  $H=2$ . We tested  $H = 1$  to 3 and noted that most of the benefits are achieved by  $H = 2$ , while  $H = 3$  did not bring noticeable improvement. We also found empirically that

changing the set of VMs does not change the above observation. We further find out via a Monte Carlo simulation that **a small  $H$  is sufficient for reaping most benefits of proxy hedging**. It is presented in Appendix C that corroborates: (i) latency and its variance reduces as  $H$  increases, and (ii) increasing  $H$  shows diminishing returns.

### 4.3 Receiver Hedging

If a VM belonging to an MP becomes a straggler, that MP would be at the losing end of many trades, as the market data it receives may lag behind other MPs. As a remedy, we assign each MP two receiver VMs, where both may execute the same trading program, a deterministic state machine. Since anomalies are less likely to affect both VMs simultaneously, performance of a trader improves significantly at the tail.

As duplicate orders will be generated by the two VMs, they can be de-duplicated by the exchange using a hash carried by each order which is computed to be the same on identical orders across the two VMs (Appendix E). Furthermore, a simple mechanism where each VM processes the packets in order of their sequence numbers (while requesting retransmission for lost packets from a rewriter [25], a common practice in the exchanges) is sufficient for ensuring that **the states of two VMs never diverge under packet losses**. We explain this further in Appendix E and provide a proof showing how packet losses can only make the state of one VM lag behind the other VM but will never diverge and non-identical orders will never be issued.

While Appendix E outlines a mechanism for maintaining synchronized VM states, the synchronization is offered as an optional feature. For some trading strategies, the benefits of prompt responsiveness may outweigh the costs of state divergence. Therefore, some MPs may opt out in favor of responding quickly to out-of-order received market data.

### 4.4 Remarks on Multicast Packet Losses

Packet losses adversely impact fairness. They lead to different view of the market for different MPs. On-premises' exchange infrastructure is engineered to provide negligible packet losses and the MPs only have to request a retransmission rarely. Onyx multicast service needs to similarly provide low losses. Our redundancy techniques help lower the packet losses, but fortunately even without our techniques, the packet losses in the cloud are very low.

GCP's Global Performance Dashboard reports 0.00262% packet-loss in the us-east4-c region for all VM-to-VM communication over a 7-day period, from April 25, 2025, 2:33 PM to May 2, 2025, 2:00 PM.<sup>5</sup> During this period, the peak loss rate (per minute) stayed at  $\leq 0.0174\%$ . We attribute such small packet losses to the fact that cloud VMs come with assigned egress and ingress capacity and as long as VMs stay within their capacity budget, they do not experience substantial packet losses. We design Onyx accordingly.

We also perform an experiment to study the losses. During our 1-hour benchmark with 100 participant VMs on GCP, the overall packet losses stayed below 0.005%, while the peak packet losses (i.e., losses observed in any one minute duration) never exceeded

0.045%, which implies that the losses are also small at tail. Utilizing receiver hedging further decreases the losses to negligible rates (more details in Appendix F).

## 5 ORDERS SUBMISSION SERVICE

We aim to provide an order submission service that can achieve both inbound fairness and high throughput. We introduce a sequencer at the exchange server to enable fair processing of orders. We install a novel scheduling policy, Limit Order Queue (LOQ) policy, on each gateway and proxy, which significantly improves the matching rate of the exchange. For simplicity, we first describe the sequencer and LOQ in a setting where gateways are directly connected to the exchange server (without a tree). Then, we explain how the multicast tree is integrated to improve the performance of the order submission service where the sequencer and LOQ are employed at each tree node to help maintain inbound fairness while handling bursts of orders.

### 5.1 Sequencer

The exchange has to provide a fair chance of trading to all the MPs, which requires the exchange to process MPs' orders following their generation timestamps (attached by trusted gateways). On-prem exchanges equalize the latency between MP servers and the exchange server by connecting them with the cables of the same length. Such an approach is not feasible in the public cloud where the tenants have no access to the underlying infrastructure. We seek an alternative approach by using synchronized clocks, widely available nowadays.

We synchronize the clocks on both MPs and the exchange VMs with an accuracy of 10s of nanoseconds using Huygens [10],<sup>6</sup> a clock synchronization algorithm robust to latency variance. We place a sequencer at the exchange server, which holds the incoming orders and releases them according to the global FIFO order i.e., the sequencer ensures that an exchange sees the orders with timestamps in the non-decreasing order. The sequencer only releases an order of an MP to the ME when (i) it sees higher timestamped orders from every other MP and the (ii) the current order has the lowest timestamp among the orders present at the sequencer.<sup>7</sup> For liveness (i.e., sequencer does not block processing of some orders for long periods), gateways periodically generate dummy orders on behalf of inactive MPs. Given ordered delivery per MP (e.g., via TCP), the sequencer ensures *safety*, i.e., inbound fairness. This is different from the prior works (§7): (i) CloudEx violates safety even with accurate clock synchronization because it waits for orders only until a set timeout and, (ii) DBO holds safety, although always, for only one class of orders that depend on last received market data. Onyx's safety comes at the cost of liveness as a failure of a gateway will block the sequencer. As is common practice in distributed systems [26–29], the failure detection of gateways is conducted by a standalone *configuration manager* (e.g., Zookeeper), and is beyond the scope of this work.

**5.1.1 Sequencer Mechanism.** MPs generate order messages that are sent to the exchange server via a reliable channel that

<sup>5</sup>GCP measures this by sending and receiving probe packets between VMs, not utilizing the actual clients' traffic.

<sup>6</sup>As we deal with latencies on the order of microseconds, such a clock synchronization accuracy is sufficient.

<sup>7</sup>Ties can be broken by any mechanism made public to MPs (e.g., MP ID).

provides in-order delivery (e.g., TCP). The exchange maintains a limit order book (LOB) and runs continuous <price, time> priority matching algorithm [12] on all the incoming trade orders. We use  $m_i^t$  to represent an order message from  $p_i$  (i.e.,  $i^{th}$  MP) with an order generation timestamp  $t$ . At the exchange server, messages are fed to the sequencer. Sequencer works in a streaming fashion, takes the messages as input, sequences them and releases them to the ME.

The sequencer maintains  $q : \{m_i^t\}$  where  $q$  is a priority queue with lexicographic ordering on  $\langle t, i \rangle$  of messages. The sequencer supports ENQUEUE and DEQUEUE operations. Pseudocode is presented in Appendix M.

**ENQUEUE:** For each new message, (i) it is added to  $q$  and, (ii) de-queue is invoked repeatedly as long as  $q$  contains non-zero number of messages from each  $p_i$ .

**DEQUEUE:** One message  $m_i^t$  is dequeued from  $q$ .  $m_i^t$  is considered sequenced at this point and presented to the ME.

## 5.2 Limit Order Queue

During periods of bursty activity in the market, incast at the exchange's ingress occurs, leading to excessive TCP retransmissions and queue build-ups at the gateways. To tackle the queuing delays incurred at the gateways, we develop a scheduling scheme—Limit Order Queue (LOQ) scheduling—to schedule the orders at each gateway, which can effectively reduce order matching latency and improve the order matching rate at ME during periods of bursts. LOQ is an application-level priority queue which is serviced in a work-conserving manner. It attempts to schedule orders in a way to enhance ME's performance while preserving inbound fairness.

**5.2.1 LOQ Mechanism.** LOQ running at each gateway (and at proxy VMs as described later) takes the orders as input and schedules them to keep the matching engine (ME) busy; ME idles if it receives orders that cannot be matched and they just need to wait in the limit order book. LOQ categorizes the orders into two classes: orders with prices closer to the mid-price, referred to as *critical orders*, are matched by the engine before other *non-critical orders*, which remain in the limit order book (LOB) awaiting favorable mid-price movement. LOQ leverages this domain knowledge to identify and prioritize critical orders over non-critical orders so that the exchange does not waste time in receiving orders that are not going to get matched soon and only needs to be put in the limit order book while the critical orders remain to be processed. In the following we explain, how LOQ works and how mid-price movement is accounted for.

LOQ scheduling, for preserving fairness, requires two parameters: (i) mid-price  $m$  and, (ii) action window  $w$ . As gateways receive all the market data, they have enough information to infer  $m$  (with a lag proportional to the multicast latency). By design of our outbound communication, gateways receive each market datum at the same time (with high probability) so they infer  $m$  almost simultaneously. Simultaneous inference is an assumption for LOQ to preserve inbound fairness, and as sometimes this assumption can be violated, we later study its impact on fairness. The action window  $w$  indicates *irrationality tolerance*: (i) if an asset is available for purchase for a price  $m$  then a buyer can bid on it with a maximum price of  $m + w$  and, (ii) if an asset has a highest bid of price  $m$  then

a seller can ask for a minimum price of  $m - w$ .  $w$  is configured by the exchange operator.

An incoming order is categorized as critical if it has a price in the range  $[m - w, m + w]$ , otherwise it is non-critical i.e., if a bid has a price in the range  $(-\infty, m - w)$  or an ask has a price in the range  $(m + w, \infty)$ , then it is non-critical.

An LOQ is a priority queue that sorts the orders lexicographically by tuple  $\langle I_m, c, t \rangle$  where  $I_m$  starts from 0 and is incremented every time  $m$  changes,  $c$  is 0 if an order is critical otherwise it is 1 and  $t$  denotes the generation timestamp of an order. LOQ maintains a global variable  $I_m$  while  $c$  is calculated per order while enqueueing it.  $I_m$  at different gateways may sometimes differ because of non-perfect simultaneous delivery and packet losses which has an adverse impact on inbound fairness discussed later. LOQ design is motivated by the need to preserve inbound fairness most of the time i.e., as long as assumptions of clock synchronization and simultaneous delivery hold.

**5.2.2 Inbound fairness under LOQ.** For safety, the sequencer assumes that messages of a single MP arrive at the sequencer in their generation order. As LOQ at a gateway reorders the messages, this assumption is violated. However, we claim that fairness is still achieved as we have designed the LOQ policy to do so. Here we provide the intuition of our claim while Appendix I further expands on it.

Assume there is a static mid-price. Instead of looking at the order in which orders are seen by the ME, let's look at the order in which the orders are executed by the ME. Executable/critical orders are those with price in  $[m - w, m + w]$  i.e., they get executed strictly before the other orders. LOQ ensures that if an order is selected for execution by ME, then all the older executable orders must have been executed already as LOQ ensures that executable orders are sorted by their timestamps (and a sequencer consults such LOQ instances and sequences them in order of timestamps). So, ME never executes an executable order if it has not executed all the older executable orders which preserves fairness. When the mid-price moves, LOQ ensures that the orders generated after the movement have lower priority than the prior orders (due to  $I_m$  being the first element of the tuple used by LOQ's priority function), leading to fairness.

**5.2.3 Impact of multicast losses on LOQ.** If a multicast packet carrying information that could update an MP's mid-price view is lost, the mid-price ID of that MP will lag behind those of other MPs. In such cases, unfairness will persist until the lost packet is recovered (or the mid-price ID is refreshed by any future received message). This underscores the importance of maintaining low packet losses, an objective met by today's public cloud infrastructure (§4.4). We later evaluate the impact of losses on LOQ via a simulation.

## 5.3 Reusing Multicast Tree In Reverse

As the number of MPs grows, the exchange's performance degrades as it cannot keep up with offered load. We find that even if the *average* offered load to an exchange stays constant, the growing number of MPs leads to performance degradation as *instantaneous*

load may exceed the exchange’s capacity due to increased fan-in factor.

In such scenarios, we observe a significant losses of MPs’ orders that leads to decrease in the exchange’s throughput and increased latency for orders. We employ a simple strategy: reuse the multicast tree in the reverse direction for relaying the orders to the exchange. The reduced fan-in at the exchange reduces the losses while packet queues form at the tree nodes. We run LOQ at each tree node to service the queues, leading to a significantly higher throughput.

Furthermore, running LOQ at proxy nodes leads to better scheduling compared to running it at just the gateways. A proxy has messages from several MPs in its queue and can prioritize messages of one MP over the messages of another MP –as long as it does not affect fairness– whereas an LOQ at a gateway only has the opportunity of prioritizing messages of an MP over the messages of the same MP.

**Fairness when using a tree.** Achieving fairness requires that an LOQ instance at each tree node is accompanied by a sequencer instance so that LOQ can operate on the messages of all children nodes fairly. Without a sequencer, a delayed message of an MP may not get assigned its proper priority by LOQ, which would eventually result into safety (fairness) violation at the exchange server. When a sequencer is used at every tree node, it only accounts for messages from its children nodes and not all MPs.

**Other methods for dealing with incast:** Several methods (Homa [30], Protego [31], Breakwater [32], DCTCP [33]) for incast mitigation and overload control have been proposed in the literature. The methods are largely orthogonal to our technique of utilizing the tree in reverse which we can do solely because we have the control of an application that lends us a tree of VMs. Credit-based overload control schemes [31, 32] are composable to Onyx, enhancing Onyx’s performance by reducing packet losses/retransmissions. However, a tree further gives us the opportunity to do better scheduling (LOQ) on multiple clients (MPs) data which is not possible without a tree/intermediate nodes and forming the queues only at the order gateways; as would be the case if we use existing techniques.

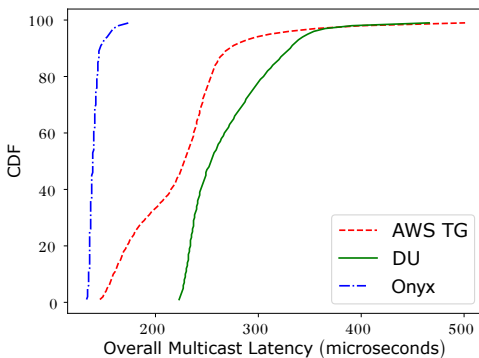


Fig. 5: Onyx has lower OML than DU, AWS TGW

## 6 EVALUATION

We focus on comparison against a prior system, CloudEx [2]. As Onyx can be viewed as CloudEx augmented with techniques to enhance performance under large number of participants, the comparison with CloudEx shows how far CloudEx can be scaled. We also present an ablation study as Onyx.

For most of the experiments, we use 100 MPs and a 5K multicast messages per second (MPS) rate. For scale, some experiments utilize 1000 MPs and this is mentioned in the respective sections. In our experiments, no two MPs share a VM. In practice, multiple MPs could be hosted in one VM to further enhance scalability. As receiver hedging improves performance at the cost of one extra VM per MP, we do not include it in most experiments to better exhibit the impact of other techniques. It is mentioned wherever it is used.

Overall multicast latency (OML) refers to the latency experienced by the last receiver that receives a multicast message. The delivery window size (DWS) is the maximum difference in the latency of any two receivers. One DWS sample is computed for each multicast message, so any mentioned percentiles (50p DWS) correspond to the aggregated statistic over the messages. We use Huygens algorithm [10] that synchronizes clocks of all the VMs with a 90p offset of  $\leq 100$  ns.

### 6.1 Multicast Latency Comparison

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

- Direct Unicast (DU): Sender directly sends a copy of a multicast message to each receiver. `io-uring` is utilized to minimize the overhead of syscalls and we observe it has better performance than socket based DU, so we utilize `io-uring` based DU for comparing against Onyx.
- AWS Transit Gateway: AWS provides TGW-based multicast [1]. It requires the sender to send message to a gateway which then replicates and sends one copy to each receiver. It can support at most 100 receivers [34] per multicast group.

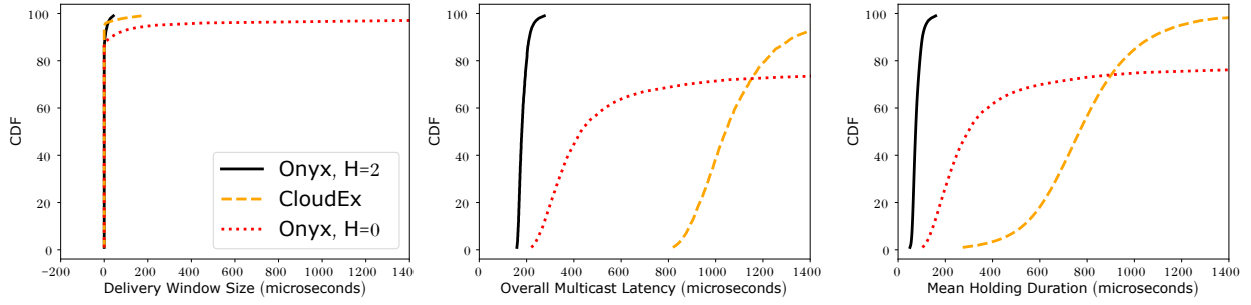
Figure 5 shows that Onyx outperforms DU and AWS-TG. The median latency for Onyx is  $129\mu\text{s}$  which is 43% lower than the latency of AWS TG ( $228\mu\text{s}$ ) and 49% lower than the latency by DU ( $254\mu\text{s}$ ). At 90p, Onyx shows  $\approx 75\%$  lower latency than both DU and AWS TG. Moreover, Onyx shows predictable latency as higher percentiles are very close to the median in contrast to the other techniques.

### 6.2 Outbound Fairness Comparison

We measure the DWS of Onyx and CloudEx to compare their outbound fairness. A smaller DWS indicates a higher level of outbound fairness (i.e., simultaneous delivery).

Figure 6a shows that Onyx achieves a DWS of  $\leq 1\mu\text{s}$  at very high percentiles (up to 92p). Without proxy hedging, the DWS becomes larger than  $\leq 1\mu\text{s}$  at earlier percentile (87<sup>th</sup>). CloudEx achieves fairness but at the cost of high OML. Later, we show that using receiver hedging achieves a DWS of  $\leq 1\mu\text{s}$  at 99.9p for Onyx.

Figure 6b shows that OML increases significantly for CloudEx and Onyx without proxy hedging. In these systems, the deadlines calculated by the hold-and-release are far into the future to cover



(a) Onyx with hedging achieves a narrow DWS (b) CloudEx shows high OML, like Onyx  $H = 0$  (c) Onyx with hedging requires less holding

Fig. 6: Outbound fairness and comparison with CloudEx

the high latency variance. This leads to high holding duration at the receivers as shown in figure 6c, increasing OML.

### 6.3 Scaling Onyx Multicast

**Scaling  $N$  to 1000.** In contrast to the previous fair multicast solutions that only work with a few 10s of receivers (e.g., [1, 4, 6]), Onyx aims to implement a more scalable multicast service which can support 1000 receivers and potentially even more. Figure 7 plots Onyx’s multicast latency for 100 receivers and 1000 receivers. Our overlay techniques for reducing latency enable Onyx to keep a graceful latency growth as the number of receivers increase.

**Fairness.** When employing hold-and-release, a median DWS of  $\leq 1 \mu s$  is achieved for  $N = 1000$ . However, the highest percentile at which DWS stays  $\leq 1 \mu s$  decreases as  $N$  increases. We define the probability of fairness ( $P(F)$ ) as the highest percentile where DWS is  $\leq 1 \mu s$ . We achieve  $P(F) = 92$  for 100 receivers and  $P(F) = 89$  for 1K receivers. We will show later that receiver hedging increases  $P(F)$  to 99.9 at the cost of one extra VM per receiver.

**Throughput.** With 466B packets, we achieve a multicast message rate of 350K MPS, without proxy hedging (and 175 MPS with proxy hedging,  $H=1$ ). Onyx utilizes  $\approx 80\%$  of the egress bandwidth of a c2d-highcpu-8 VM on GCP while maintaining negligible multicast packet losses. The specified VM has an egress bandwidth of 16Gbps.

### 6.4 Orders Submission Performance

Figure 8 shows the order matching rate of the exchange server. Onyx outperforms CloudEx by an order of magnitude. 100 MPs cumulatively generate 100K orders per second each where periodic bursts occur (shown as shaded regions) increasing the order generation rate 20x. MPs stop generating the orders after 20s. Onyx is able to keep up with the offered load and finishes processing the orders right after the MPs stop. CloudEx achieves significantly low order matching rate and builds up queues that are processed long after the MPs stop. Onyx achieves higher order matching rate when bursts occurs due to LOQ. The median latency of orders with Onyx is  $\approx 97\%$  lower compared to CloudEx.

**Limit Order Queue (LOQ) Performance** We compare LOQ to a FIFO queue, which we refer to as SimplePQ. SimplePQ does not differentiate between critical and non-critical orders, but sorts any

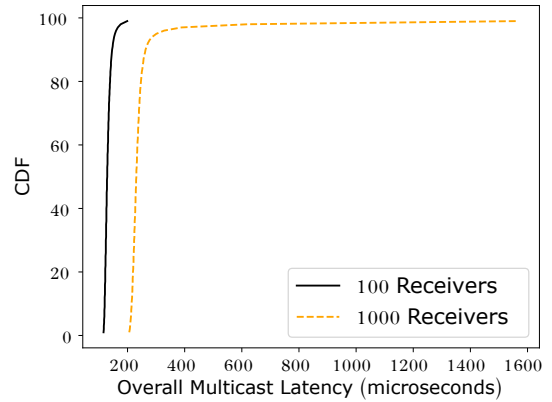


Fig. 7: Onyx scales well with median OML  $\leq 250\mu s$

queued orders by timestamps while LOQ sorts by criticality as well as by timestamps. As sorting by timestamp helps the ME which needs to sort the order similarly, we isolate the effect of LOQ’s sorting by criticality by having SimplePQ sort the orders by timestamp as well.

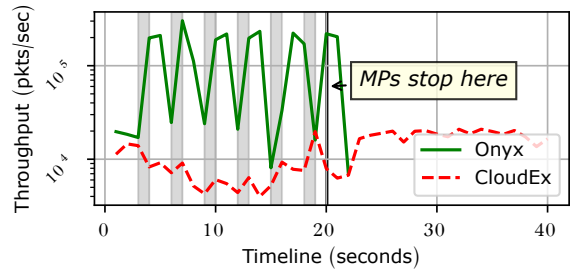
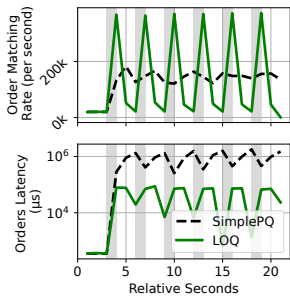
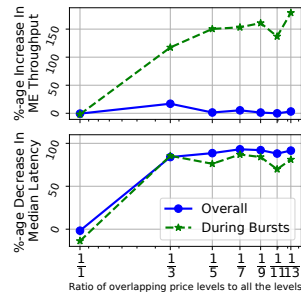


Fig. 8: Onyx achieves high order matching rate.



**Fig. 9: LOQ effectively handles bursts (shaded).**



**Fig. 10: Benefits are proportional to book depth.**

In this experiment, we have 10 market participants (MPs), each submitting 10k orders per second. Two intermediate proxies, running LOQ or SimplePQ, relay orders from the MPs to the ME. Only queue type (LOQ/SimplePQ) changes for performing a comparative experiment. Order bursts occur, with each MP increasing their order submission rate by 20× every 3 s. Bursts lead to queue build-ups at the proxies, where LOQ/SimplePQ operate. As shown in Figure 9, unlike SimplePQ, LOQ gracefully handles bursts: it achieves higher order matching and lower latency during bursts and leads to low latency overall. Order matching rate of LOQ goes below that of SimplePQ in-between the bursts. This behavior is expected because LOQ is able to process a burst *as it occurs* while SimplePQ queues up orders and continues processing them after the burst vanishes, increasing latencies.

The advantages of LOQ stem from the presence of non-critical orders, which are kept in the gateways and proxies’ queues longer, allowing the available resources (network bandwidth and ME’s capacity) to be used for critical orders. We examine the benefits of LOQ under different ratios of critical to non-critical orders. In this experiment, MPs uniformly sample bid/ask prices from a predefined range to generate orders. We vary the portion of the bid price range that overlaps with the ask price range. These overlapping segments result in orders that get matched at ME, while other orders remain in the LOB. Figure 10 shows that, as the ratio of overlapping price levels to the total number of levels decreases, the benefit of LOQ increases. A ratio of  $x/y$  indicates that there are  $y$  total price levels from which MPs sample prices to generate orders, but only  $x$  price levels lead to immediate matches at ME. As  $y$  increases while  $x$  remains constant, the number of critical orders decreases. We also plot the metrics for the bursts durations. These are calculated from the orders matched during the bursts. LOQ identifies critical orders and prioritizes them for processing by the ME over non-criticals. Consequently, LOQ demonstrates a 137% increase in ME throughput during bursts and a 70% reduction in latency for matched orders when  $y = 11$ . We repeat one experiment and increase MPs from 10 to 1000 with  $x = 1, y = 7$  and observe that LOQ outperforms SimplePQ: 90% decrease in the overall latency and 85% increase in matching rate during bursts.

**Using Proxy Tree for Orders Submission:** To conduct an ablation study of the proxy tree’s benefit in the order submission, we run

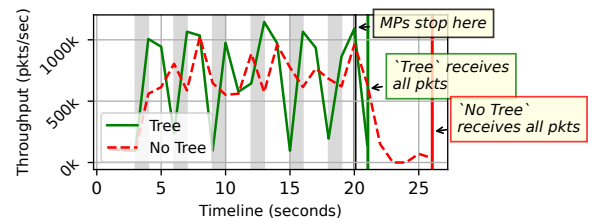
an experiment without LOQ and sequencer, and measure the number of packets received per second by the exchange server with and without a tree. Figure 11 plots the throughput achieved with 100 MPs. 100 MPs send at the cumulative base rate of 100K packets/second, but create periodic bursts that reach 20× higher than the base rate. By using the proxy tree, Onyx can improve the overall throughput by ~ 22% on average and by ~ 75% during the bursty periods.

**Sequencing Overhead:** We perform an experiment with 100 MPs (without LOQ) to study the throughput with and without our sequencer. We use the same load generator as above and notice an overall 25% decrease in the throughput of the exchange when using the sequencer. The throughput is essentially traded-off for improving inbound fairness by a sequencer. The overhead comes from the fact that for releasing any single message of a downstream node, the sequencer has to wait for at-least one higher timestamped message from every other downstream node. The speed of the entire system becomes dependent on the speed of the slowest messages, a pattern common in many fairness preserving systems [6]. Appendix J further discusses the overhead and presents potential ways to alleviate it e.g., by using several instances of a sequencer so that a path to at least one instance has high chances of being free of latency fluctuations.

**Impact of Packet Losses On LOQ:** When an order gateway has a stale mid-price due to packet losses, two cases may happen: (i) corresponding market participant (MP) does not generate orders until losses are recovered or (ii) orders are generated based on the stale information. In the first case, there is no impact of losses on LOQ or on inbound fairness. However, a type of unfairness exists in the systems as some MPs are not able to generate orders. This unfairness is irrespective of LOQ and exists in Onyx as well as all prior systems including on-premises exchanges.

In the second case, the generated orders by the MPs who have stale mid-price (and hence, a lower mid-price ID in the corresponding order gateway) will get prioritized over the orders of other MPs who have the latest mid-price (because the LOQ prioritization tuple has the mid-price ID as the first element). We study this impact quantitatively via a simulator.

A simulator allows us to keep all the conditions (loss rate and message generation timestamps) identical while performing comparative experiments i.e., comparing the behavior when employing LOQ vs. a FIFO. With losses, the output sequence of matched orders may change while employing LOQ as compared to employing a FIFO. We measure how much change may appear. Output sequence



**Fig. 11: A tree enhances an exchange’s throughput.**

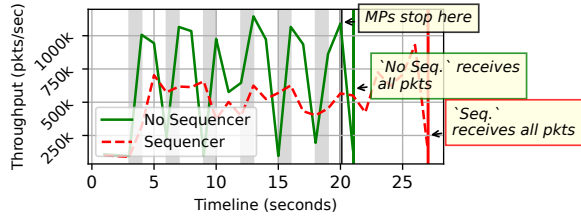


Fig. 12: Sequencer has a significant overhead.

when using FIFO is called *fifo-seq*, while the one with LOQ is called *loq-seq*. We define *lateness*: an order appearing at index  $i$  in *fifo-seq* and at index  $j$  in the *loq-seq* has the lateness of  $|i - j|$ . We measure lateness for each order via our simulator. Lateness of 0 is ideal. Higher lateness represents lower fairness.

The simulation utilizes 100 participants (MPs), each submitting 5K orders per second. The exchange adds mid-price ID to the outgoing messages so an MP with lost packets and hence, stale mid-price ID will recover the ID once it receives any future message. Figure 13 shows that lateness is proportional to the packet loss rates. For the usually observed losses ( $\leq 0.005\%$ ), the median lateness is 0, while 90p stays  $\leq 3$ . More details are in Appendix N showing: (i) higher loss rates lead to higher lateness and, (ii) fewer the clients experiencing the losses, lower the lateness. The results convey that small losses (which are typical of the public cloud) do not introduce unfairness to a vast majority of the orders. The impact on the eventual trading outcomes is specific to the trading strategies employed by the MPs, we only exhibit here that lateness, although small, may exist under packet losses.

**Remarks on inbound fairness:** By design, our sequencer (w/o LOQ) ensures inbound fairness. One of the assumptions for LOQ to preserve fairness is that all MPs/gateways infer the mid-price movement simultaneously which is provided with a high probability. Although losses impact inbound fairness, non-simultaneous delivery has a bigger impact. The probability of achieving inbound fairness while employing LOQ can only be as high as that of outbound fairness (simultaneous delivery) which is optimized to 99.9% (§6.5.3). With receiver hedging,  $\approx 0.1\%$  packets lead to non-simultaneous delivery (§6.5.3) i.e. these packets are delayed enough that hold-and-release protocol does not meet its deadlines. However, figure 13 shows that even assuming 0.1% losses (i.e., considering all delayed packets as lost), the impact is small.

## 6.5 Outbound Communication Techniques

### 6.5.1 Proxy Hedging.

**Reduced overall multicast latency.** Figure 14a compares the OML CDFs under different hedging factors ( $H = 0, 1, 2$ ).  $H = 0$  represents the case without proxy hedging.  $H = 1$  yields latency reduction as the CDF curve is shifted to the left. The latency reduction is marginal as  $H$  increases from 1 to 2, as increasing  $H$  has diminishing returns.

**Reduced temporal and spatial latency variance.** We calculate the median OML over a tumbling window of 5K messages to study the temporal latency variance. Figure 14b shows Onyx exhibits lower

temporal latency variance compared with the non-hedging scheme ( $H = 0$ ). Enhancing temporal predictability, even for short time-steps, is beneficial for hold-and-release to calculate lower deadlines.

Figure 14c compares the DWS for Onyx with different hedging factors. Hedging reduces the spatial latency variance (i.e., it shrinks the DWS of multicast messages). The 99th percentile delivery window size is  $\sim 350 \mu s$  with  $H = 0$  but  $\sim 150 \mu s$  with  $H = 1$ , and slightly better with  $H = 2$ .

Appendix H demonstrates (i) temporal latency experienced by a receiver decreases because of proxy hedging and, (ii) the dollar cost of proxy hedging is negligible compared to the costs borne by on-premises exchanges.

**6.5.2 Round Robin Packet Spraying.** We evaluate the impact of RRPS on OML. We utilize a multicast message rate of 10K MPS, 125 receivers and a burst of 1 s occur every 2 s, increasing the MPS 15x. Table 1 shows multicast latency decreases when RRPS is used. On GCP, we see an OML reduction of  $\approx 10\%$ , however, on AWS the reduction can approach  $\approx 70\%$ . With low message rates, the reduction is not significant:  $\leq 5\%$  with message rates  $\leq 100K$  MPS. This happens because the inter-packet duration is large enough that any transient latency spike is not able to impact consecutive messages so RRPS does not help much by re-routing messages.

RRPS show improvements in OML when message bursts are introduced as it can distribute the bursts among several network paths. Without bursts, the OML reduction still happens but is less than 10% on both AWS and GCP.

**6.5.3 Receiver Hedging.** Receiver hedging improves OML (Table 2) and achieves outbound fairness/simultaneous delivery (i.e.,  $DWS \leq 1 \mu s$  at 99.9p). This technique improves performance at the cost of one extra VM per MP. Doubling the amount of receiver VMs may also increase the number of proxies in the tree as  $< D, F >$  is appropriately tuned.

Setup	On AWS		On GCP	
	50p	90p	50p	90p
Improve. (%)	15.12	69.35	9.83	9.18

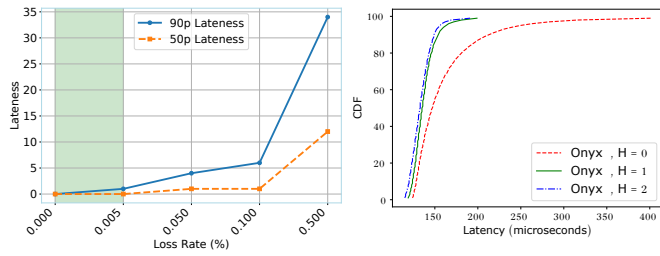
Tbl. 1: %-age improvement in OML due to RRPS

N	RH	OML ( $\mu s$ )	
		50p	99p
100	No	139	248
200	No	141	243
100	Yes	99	146

Tbl. 2: Latency impact due to Receiver Hedging (GCP)

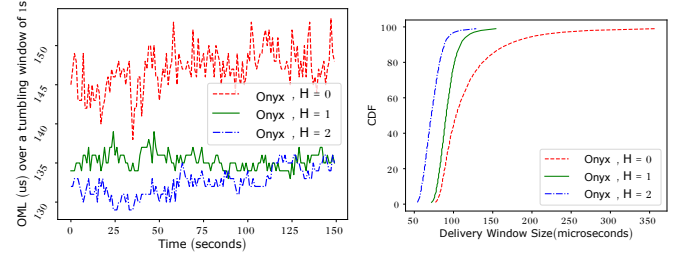
## 6.6 Onyx and DBO

DBO is a recent cloud exchange that achieves fairness regardless of latency fluctuations and without clock synchronization. It does so by proposing a new fairness metric that prioritizes orders based on the response time of traders (i.e., the time to make a trade in



**Fig. 13: Low latency with low loss rates.** (simulation)

**(a) Hedging reduces OML**



**(b) Low temporal variance**

**(c) Hedging reduces DWS**

**Fig. 14: Evaluating Proxy Hedging**

response to a piece of market data), rather than the time at which orders were submitted. Onyx can be viewed as a network layer that is complementary to DBO’s new semantics at the application layer. In particular, Onyx can help DBO scale out to a large number of participants in the following three ways. One, Onyx’s low-latency and scalable multicast tree (and *hedging*) can help with disseminating market data in DBO. Two, Onyx’s use of a tree in reverse for inbound order submission can also help DBO deal with incast-type conditions during order submissions from a large number of participants. Three, because DBO does not enforce simultaneous delivery of data, DBO has an extra constraint that no MPs should directly or indirectly talk to each other so that an MP with earlier received data may not leak it to any other MP as it can violate DBO’s fairness guarantees. By leveraging Onyx’s simultaneous multicast in DBO, this constraint can be lifted. Appendix P details its order submission service’s performance which is lower than Onyx and CloudEx.

## 7 RELATED WORK

We have already discussed the recent exchanges (CloudEx [2], DBO [6]) and overload control schemes [30–32].

**Multicast:** Prior works on application level multicast [21, 22, 35–37] mainly focus on finding optimal paths in a network (or overlay mesh) using cost models for network *links* that capture heterogeneous link bandwidth or latency characteristics. Onyx focuses on achieving a small latency difference across receivers while balancing transmission and propagation delays via a tree. Switch-based multicast [38, 39] is not available to cloud tenants due to scalability issues [40]. Some clouds provide special offerings for multicast [1] that is implemented using multiple unicasts and lack performance.

**Collective Communication:** Collective communication [41–43] uses overlay trees for broadcast and all-reduce [44], but typically supports fewer than 100 receivers. For example, Hoplite [41] optimizes bandwidth for 10s of nodes using a tree. Onyx focuses on minimizing latency and variance, handling bursts, and achieving fairness at scale while not having an *aggregation* mechanism for trade orders that could enhance performance via an *all-reduce* mechanism.

## 8 LIMITATIONS & FUTURE WORK

**Coarse Fairness Guarantees:** If all MPs receive a message within 1  $\mu$ s of each other, we consider it fairly disseminated. However, in on-premises exchanges, fairness guarantees are more precise, down

to the level of nanoseconds. Achieving such fairness with Onyx requires synchronizing the clocks of all receiver VMs with accuracy better than nanoseconds – a level of precision that current public cloud technology does not yet provide.

**High Order Latencies:** As LOQ enables graceful handling of bursts of orders by keeping the latencies lower than the alternative of using FIFO queues at proxies, it still has much room for improvement. The latencies of trade orders, due to bursts, can reach 10s of milliseconds which is significantly higher than the latencies on the outbound side ( $\approx 100 \mu$ s). To keep the order latencies bounded, there could be admission control applied at the order gateways.

**LOQ:** All orders generated before a mid-price movement gets processed before all the orders generated after the mid-price movement as LOQ has the mid-price ID as the first element in the prioritization tuple. This is designed to retain inbound fairness, but it trades-off some potential performance gains. The direction of mid-price movement (left or right) can be taken into account to decide whether some older orders (with lower mid-price ID) can be de-prioritized over some newer orders (with high mid-price ID). We have not explored it and leave it as a future work.

**Opportunity to utilize Cloud FPGAs:** In on-premises exchanges, some MPs implement their trading algorithms on FPGA to achieve low-latency processing. An interesting question is whether FPGA instances in the cloud [45] can serve the same purpose. In AWS, FPGAs do not have direct network access; instead, an associated VM receives the packets and then forwards them to the FPGA. This raises uncertainty about whether FPGA-based low-latency processing of orders is feasible in such an environment. Similarly, whether the Onyx proxies can be offloaded to FPGAs is an interesting direction for enhancing performance.

## 9 CONCLUSION

This paper introduces Onyx, which provides networking support for scalable cloud financial exchanges. Onyx systematically optimizes both outbound (market data delivery) and inbound (order submission) workflows of an exchange system. Our evaluation shows that Onyx outperforms a prior system CloudEx, in terms of fairness, throughput, and latency as well as outperforms AWS TGW-based multicast. This work does not raise any ethical issues.

## 10 ACKNOWLEDGMENTS

We thank Deepak Merugu for help with clock synchronization agents, Balaji Prabhakar for initial discussions about cloud multi-cast, Mathew Grosvenor for helpful discussions, and Fabian Ruffly for providing feedback at various stages of the project. We thank Daniel Qian for help in writing Appendix I. This work was supported by NSF grants 2422076, 2019302, NSF CAREER award (2340748) and a gift from the eBPF foundation and Xilinx.

## REFERENCES

- [1] Amazon Web Services. Multicast on transit gateways. <https://docs.aws.amazon.com/vpc/latest/tgw/tgw-multicast-overview.html>, .
- [2] Ahmad Ghalayini, Jinkun Geng, Vighnesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Cloudex: A fair-access financial exchange in the cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '21*, page 96–103, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384384. doi: 10.1145/3458336.3465278. URL <https://doi.org/10.1145/3458336.3465278>.
- [3] leptonsys.com. Layer 1 switches: Key functions and technologies. <https://www.leptonsys.com/blog/layer-1-switches-key-functions-and-technologies>.
- [4] Junzhi Gong, Yuliang Li, Devdeep Ray, KK Yap, and Nandita Dukkkipati. Octopus: A fair packet delivery service. *arXiv preprint arXiv:2401.08126*, 2024.
- [5] Prateesh Goyal, Ilias Marinos, Eashan Gupta, Chaitanya Bandi, Alan Ross, and Ranveer Chandra. Rethinking cloud-hosted financial exchanges for response time fairness. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks, HotNets '22*, page 108–114, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398992. doi: 10.1145/3563766.3564098. URL <https://doi.org/10.1145/3563766.3564098>.
- [6] Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. Dbo: Fairness for cloud-hosted financial exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 550–563, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi: 10.1145/3603269.3604871. URL <https://doi.org/10.1145/3603269.3604871>.
- [7] Nasdaq. Nasdaq co-location services, 2024. URL <https://www.nasdaqtrader.com/TRADER.ASPX?ID=COLO>. Accessed: 2025-05-20.
- [8] Liangcheng Yu, Prateesh Goyal, Ilias Marinos, and Vincent Liu. Cutfleish: A fair, predictable execution environment for cloud-hosted financial exchanges, November 2024. URL <https://www.microsoft.com/en-us/research/publication/cutfleish-a-fair-predictable-execution-environment-for-cloud-hosted-financial-exchanges/>.
- [9] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 303–314, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585949. doi: 10.1145/1592568.1592604. URL <https://doi.org/10.1145/1592568.1592604>.
- [10] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 81–94, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-43-0.
- [11] CME Group. Connectivity options. [cmegroup.com/globex/connectivity-options.html](http://cmegroup.com/globex/connectivity-options.html). Accessed: 2024-09-18.
- [12] investopedia.com. Matching orders: What they are, how they work, and examples. <https://www.investopedia.com/terms/m/matchingorders.asp>. Accessed: 2024-09-19.
- [13] Eurex. Matching principles. <https://www.eurex.com/ex-en/trade/order-book-trading/matching-principles>. Accessed: 2025-06-25.
- [14] Owen Hilyard, Bocheng Cui, Marielle Webster, Abishek Bangalore Muralikrishna, and Aleksey Charapko. Cloudy forecast: How predictable is communication latency in the cloud? *arXiv preprint arXiv:2309.13169*, 2023.
- [15] Vighnesh Sachidananda. Scheduling and autoscaling methods for low latency applications. [https://stacks.stanford.edu/file/druid:xq718qd4043/Vig\\_thesis\\_submission-augmented.pdf](https://stacks.stanford.edu/file/druid:xq718qd4043/Vig_thesis_submission-augmented.pdf). Accessed: 2024-09-19.
- [16] CME Group. Cme group and google cloud announce new chicago area private cloud region and co-location facility for cme group's markets. [www.cmegroup.com/media-room/press-releases](http://www.cmegroup.com/media-room/press-releases). Accessed: 2024-09-11.
- [17] Alexander Osipovich. 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.
- [18] a16z. The cost of cloud, a trillion dollar paradox. <https://a16z.com/the-cost-of-cloud-a-trillion-dollar-paradox>. Accessed: 2025-01-27.
- [19] Andy Myers, Brian Nigito, and Nate Foster. Network design considerations for trading systems. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks, HotNets '24*, page 282–289, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712722. doi: 10.1145/3696348.3696890. URL <https://doi.org/10.1145/3696348.3696890>.
- [20] Muhammad Haseeb, Jinkun Geng, Ulysses Butler, Xiyu Hao, Daniel Duclos-Cavalcanti, and Anirudh Sivaraman. Poster: Jasper, a scalable and fair multicast for financial exchanges in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*, ACM SIGCOMM Posters and Demos '24, page 36–38, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400707179. doi: 10.1145/3672202.3673728. URL <https://doi.org/10.1145/3672202.3673728>.
- [21] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Minimum-delay multicast algorithms for mesh overlays. *IEEE/ACM Transactions on Networking*, 23(3):973–986, 2015. doi: 10.1109/TNET.2014.2310735.
- [22] G.N. Rouskas and I. Baldine. Multicast routing with end-to-end delay and delay variation constraints. *IEEE Journal on Selected Areas in Communications*, 15(3): 346–356, 1997. doi: 10.1109/49.564133.
- [23] David G. Andersen, Alex C. Snoeren, and Hari Balakrishnan. Best-path vs. multi-path overlay routing. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, page 91–100, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137737. doi: 10.1145/948205.948218. URL <https://doi.org/10.1145/948205.948218>.
- [24] Vighnesh Sachidananda. Scheduling and autoscaling methods for low latency applications. [https://stacks.stanford.edu/file/druid:xq718qd4043/Vig\\_thesis\\_submission-augmented.pdf](https://stacks.stanford.edu/file/druid:xq718qd4043/Vig_thesis_submission-augmented.pdf). Accessed: 2024-01-30.
- [25] ASX. Asx trade guide to testing services. <https://www.asxonline.com/content/dam/asxonline/public/documents/asx-trade-refresh-manuals/asxtrade-guide-to-testing-services.pdf>.
- [26] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, 2009.
- [27] Inho Choi, Ellis Michael, Yunfan Li, Dan Ports, and Jialin Li. Hydra: Serialization-free network ordering for strongly consistent distributed applications. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*, 2023.
- [28] CMAK:Cluster Manager for Apache Kafka. <https://github.com/yahoo/CMAK>. Accessed: 2021-02-02.
- [29] Apache Software Foundation. Zookeeper. <https://zookeeper.apache.org>.
- [30] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674. doi: 10.1145/3230543.3230564. URL <https://doi.org/10.1145/3230543.3230564>.
- [31] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/cho-inho>.
- [32] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for  $\mu$ -scale RPCs with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/cho>.
- [33] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302012. doi: 10.1145/1851182.1851192. URL <https://doi.org/10.1145/1851182.1851192>.
- [34] Amazon Web Services. Quotas for your transit gateways. <https://docs.aws.amazon.com/vpc/latest/tgw/transit-gateway-quotas.html>, .
- [35] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. *SIGCOMM Comput. Commun. Rev.*, 32(4):205–217, aug 2002. ISSN 0146-4833. doi: 10.1145/964725.633045. URL <https://doi.org/10.1145/964725.633045>.
- [36] M. Parsa, Qing Zhu, and J.J. Garcia-Luna-Aceves. An iterative algorithm for delay-constrained minimum-cost multicasting. *IEEE/ACM Transactions on Networking*, 6(4):461–474, 1998. doi: 10.1109/90.720991.
- [37] Yang Hua Chu, S.G. Rao, S. Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, 20(8):1456–1471, 2002. doi: 10.1109/JSAC.2002.803066.
- [38] B. Prabhakar, N. McKeown, and R. Ahuja. Multicast scheduling for input-queued switches. *IEEE Journal on Selected Areas in Communications*, 15(5):855–866, 1997. doi: 10.1109/49.594847.

- [39] Ming-Huang Guo and Ruay-Shiung Chang. Multicast atm switches: survey and performance evaluation. *SIGCOMM Comput. Commun. Rev.*, 28(2):98–131, apr 1998. ISSN 0146-4833. doi: 10.1145/279345.279352. URL <https://doi.org/10.1145/279345.279352>.
- [40] Muhammad Shabbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: source routed multicast for public clouds. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19*, page 458–471, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342066. URL <https://doi.org/10.1145/3341302.3342066>.
- [41] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21*, page 641–656, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837. doi: 10.1145/3452296.3472897. URL <https://doi.org/10.1145/3452296.3472897>.
- [42] Liangyu Zhao and Arvind Krishnamurthy. Bandwidth optimal pipeline schedule for collective communication, 2023.
- [43] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Efficient direct-connect topologies for collective communications, 2023.
- [44] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, feb 2009. ISSN 0743-7315. doi: 10.1016/j.jpdc.2008.09.002. URL <https://doi.org/10.1016/j.jpdc.2008.09.002>.
- [45] AWS. Amazon ec2 f1 instances. <https://aws.amazon.com/ec2/instance-types/f1/>. Accessed: 2024-02-09.
- [46] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for Large-Scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, March 2016. USENIX Association. ISBN 978-1-931971-29-4. URL <https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman>.
- [47] Vasilios Mavroudis and Hayden Melton. Libra: Fair order-matching for electronic financial exchanges, 2019.
- [48] Piotr J. Gmytrasiewicz and Edmund H. Durfee. Decision-theoretic recursive modeling and the coordinated attack problem. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, page 88–95, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 155860250X.
- [49] Intel. Dpdk programmer’s guide: Ring library. [https://doc.dpdk.org/guides/prog\\_guide/ring\\_lib.html](https://doc.dpdk.org/guides/prog_guide/ring_lib.html). Accessed: 2024-01-31.
- [50] Jianling Wang, Vivek George, Tucker Balch, and Maria Hybinette. Stockyard: a discrete event-based stock market exchange simulator. In *Proceedings of the 2017 Winter Simulation Conference, WSC '17*. IEEE Press, 2017. ISBN 9781538634271.
- [51] Signals Jane Street and Threads. Multicast and the markets. <https://signalsandthreads.com/multicast-and-the-markets/>.
- [52] Google Cloud. View google cloud packet loss - performance dashboard. <https://cloud.google.com/network-intelligence-center/docs/performance-dashboard/how-to/view-google-cloud-packet-loss>, 2024. Accessed: 2025-05-19.
- [53] GitHub. Performance in terms of PPS. <https://github.com/amzn/amzn-drivers/issues/68>.
- [54] Philipp Winter, Ralph Giles, Moritz Schafhuber, and Hamed Haddadi. Nitriding: A tool kit for building scalable, networked, secure enclaves, 2023. URL <https://arxiv.org/abs/2206.04123>.
- [55] Legal Information Institute. Protection of nonpublic personal information. <https://www.law.cornell.edu/uscode/text/15/6801>. Accessed: 2024-09-19.
- [56] iex. The cost of exchange services. <https://finansdanmark.dk/media/mstbqp23/iex-and-market-data-cost-2019.pdf>. Accessed: 2024-09-19.
- [57] coinbase.com. Buy and sell bitcoin, ethereum and more with trust. <https://www.coinbase.com>. Accessed: 2024-09-19.
- [58] binance.com. Crypto trading platform. <https://www.binance.com>. Accessed: 2024-09-19.
- [59] Nasdaq.com. 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.

Note: Appendices are supporting material that has not been peer-reviewed.

## A 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 (almost 40) experiments with various numbers of receivers ( $N = 10, 100, 1000$ ). Table 3 displays the latencies for different configurations of  $\langle D, F \rangle$  for a given number ( $N$ ) of

receivers. We see that the latency yields the minimum 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 no longer decreases but rises up. Based on our experiments, we find fixing  $F = 10$  usually leads to a desirable  $D$  to generate a multicast tree with low latency. As a result, we establish our heuristic rule to construct the multicast tree as follows: Given  $N$  receivers, we fix  $F = 10$  and then derive  $D = \lceil \log_{10} N \rceil$  (round to the nearest integer). We also tested a more sophisticated alternative mechanism described in [46], and observe no significant performance boost compared to our heuristic.

In our testing, the variance in cloud-host VM performance [24] imposes the great challenge to differentiate the “optimal”  $D$  and  $F$  from the values generated by 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  $30 \pm 10\mu\text{s}$  and a unit increment in  $F$  adds  $2.7 \pm 0.9\mu\text{s}$  per layer.<sup>8</sup> A tree constructed using linear models based on these unit increments performs comparably to the tree constructed using our heuristic of maintaining  $F$  close to 10 and  $D = \lceil \log_{10} N \rceil$ . Here we use two examples to illustrate how our approach decides the tree structure: For  $N = 100, F = 10, D$  would be 2. For  $N = 200, F = 10, D$  would still be 2 and  $F$  would need to be adjusted accordingly to support all 200 receivers. So  $F$  would come around to be  $14 \approx 200$ .

## B SCALABLE SIMULTANEOUS DELIVERY

Financial trading needs to ensure the fairness of the competition. Fairness in data delivery [2, 4] 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 [6, 47], these variant definitions still need more research before they can be confidently adopted by the exchanges. Therefore, Onyx uses the original definition of fairness employed by on-premise financial exchanges: 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* [48]. Nevertheless, Onyx tries to empirically minimize the spatial (i.e., across receivers) variance of the latency of messages. Our hedging design (§4.2) has created favorable conditions to minimize the spatial variance. By using hedging, Onyx 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

<sup>8</sup>Using our high-performance implementation on a c5.2xlarge VM in AWS. Message size is 460B.

(a) N=10			(b) N=100			(c) N=1000		
D	F	OML	D	F	OML	D	F	OML
1	10	66	1	100	351	2	32	282
2	4	88	2	10	139	3	10	217
			3	5	141	4	6	226

**Tbl. 3:** Median values of overall multicast latency (OML), in  $\mu\text{s}$ , for different depth ( $D$ ), fanout ( $F$ ), and receivers ( $N$ ).

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 [2] but it does not scale well and leads to high end-to-end latency (§6.2). 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, Onyx leverages the accurate clock synchronization algorithm, Huygens [10], 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:  $\text{OWD}_G = \max_i(\text{OWD}_i)$  where  $\text{OWD}_i$  is the OWD estimate reported by the  $i$ -th multicast receiver.

Once an  $\text{OWD}_G$  has been calculated by the multicast sender, it attaches deadlines to all outgoing messages. A deadline is calculated by adding  $\text{OWD}_G$  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 §G, we discussed possible security mechanisms to ensure that a receiver (an MP) waits until its deadline to process a message, while it’s in the MP’s self interest to act immediately without waiting.

**Deadlines all-reduce.** In the design of Onyx’s *hold-and-release* mechanism, all the receivers need to send back the estimates of OWDs they experience to the sender so that the sender can estimate the deadlines for the subsequent messages to multicast. However, if all the receivers send their estimated OWDs directly to the sender, incast congestion occurs at the sender, leading to increased message drops. To avoid the high volume of incast traffic, we reuse the multicast tree to aggregate the OWD estimates in an all-reduce manner [44]. Specifically, each receiver periodically sends its OWD estimate to its parent proxy. Since 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) sends the max value to its parent proxy. In this way, the sender at the root calculates deadlines for messages only according to the aggregated estimates from the first proxy layer rather than all the receivers.

## C PROXY HEDGING ANALYSIS VIA MONTE CARLO

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

We use function  $L(a, b)$  to represent the latency from node  $a$  to node  $b$ . We use  $S$  to represent the root node (i.e., the sender) in Figure 2, 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):

$$L(S, P_i^n) = \min_{0 \leq j \leq H} \left\{ L(S, P_{\lfloor i/F \rfloor - j}^{n-1}) + L(P_{\lfloor i/F \rfloor - j}^{n-1}, P_i^n) \right\}$$

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

Each  $L(P_i^{n-1}, P_i^n)$  is assumed to be independent and identically distributed (IID). However, the latency is impacted by the order in which a parent proxy sends out the messages. This happens because of the replication and the transmission delay of sending messages. If the number of downstream nodes for a proxy is small, we can ignore this order in the model. Let’s assume  $L(P_i^{n-1}, P_i^n)$  to be IID. Later, we discuss the effect of non-IID latencies.

Achieving a low variance of  $L(S, P_i^n)$  would mean that the latency over time does not deviate much from the expected value, which helps to achieve consistent latency over time. It also indicates that different VMs (at the same level of the tree) in Onyx do not experience latency significantly different from each other, which narrows the gap between the maximum and minimum latency experienced among all the receivers for a multicast message.

We run a Monte Carlo simulation of  $L(S, P_i^n)$  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 15), we have three main takeaways.

**No Hedging  $\approx$  High Latency Variance:** With no hedging, the depth of a tree and latency variance are directly correlated. Figure 15a plots the CDF for  $L(S, P_i^n)$  and shows mean value  $\mu$  and standard deviation  $\sigma$  for different configurations of the multicast tree. As  $D$  grows larger, we see both  $\mu$  and  $\sigma$  increase distinctly, indicating that just a proxy tree (i.e., no hedging) suffers from more latency variance when the tree scales up.

**Hedging  $\approx$  Low Latency Variance:** Our hedging makes the correlation between  $D$  and  $\sigma$  become less significant. In Figure 15b, we can see the latency distribution becomes *narrow* (i.e., reduced  $\sigma$ ) as  $H$  grows from 0 to higher values.

**Low Overall Latency & Diminishing Returns on  $H$ :** In Figure 15c, we fix  $D$  and keep increasing  $H$ . Figure 15c shows that

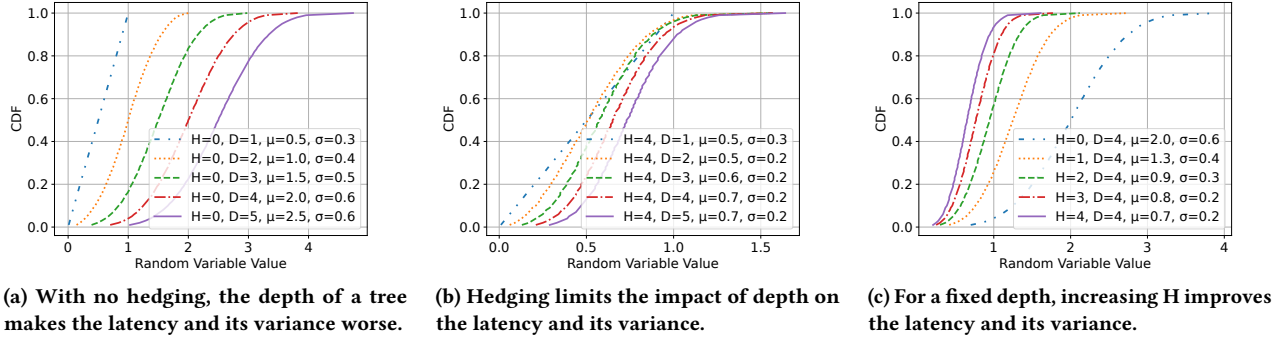


Fig. 15: Analyzing VM Hedging. A Monte Carlo simulation with 100k iterations was used.

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.

**Non-IID Latencies:** When the latencies of different links depend on one another, proxy hedging still reduces latency effectively. However, the reduction effect is inversely proportional to how much links' latencies depend on each other.

We define *direct links* as links that go from parents to their own children and *hedging links* as links that go from aunts to their nieces. For a given node, we have outgoing hedging links' latencies depend on a single outgoing direct link's latency. Assuming  $C$  is the set of a proxy  $p$ 's children,  $E$  is the set of  $p$ 's nieces, formally, we have:

$$\begin{aligned} L(P_p^n, P_c^{n+1}) &\sim \mathcal{U}, \text{ i.i.d for } c \in C \\ \varepsilon(P_p^n, P_e^{n+1}) &\sim \mathcal{U}, \text{ i.i.d for } e \in E \\ L(P_p^n, P_e^{n+1}) &= g\left(L(P_p^n, P_c^{n+1}), \varepsilon(P_p^n, P_e^{n+1})\right) \end{aligned}$$

where  $\varepsilon(P_p^n, P_e^{n+1})$  is the base latency of link from  $p$  to  $e$  and  $g$  is a deterministic transformation function. For simplicity, we model  $g$  as a linear function. Thus, we have:

$$L(P_p^n, P_e^{n+1}) = \alpha L(P_p^n, P_c^{n+1}) + \varepsilon(P_p^n, P_e^{n+1})$$

where  $\alpha$  represents the dependency factor: how correlated the hedging links are with the direct link.

We first analyzed (by running a Monte-Carlo simulation with the random variables above) how dependency affects the overall performance with a similar setup as the IID case. In Figure 16, as the correlation among links ( $\alpha$ ) increases, the hedging links get more affected by the direct link, we can see that the overall performance decreases. However, the benefit of hedging still exists even with highly correlated links ( $\alpha = 800\%$ ). As  $\alpha \rightarrow \infty$ , the overall latency approaches that of  $H = 0$  i.e., the latency benefit vanishes.

Next we experimented with how different  $H$  performs under a fixed dependency factor of 100%. In Figure 17, increasing  $H$  decreases latency. Overall, the trend is consistent with the IID case, but the absolute benefits diminish.

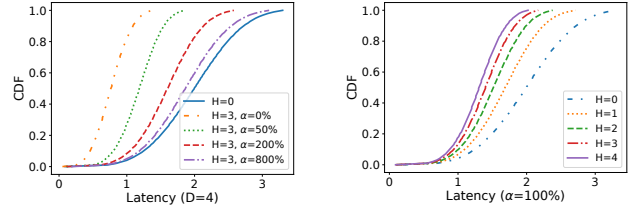


Fig. 16: Benefits diminish as links' correlation ( $\alpha$ ) increases. Fig. 17: With a fixed  $\alpha$ , increasing  $H$  improves the performance.

## D OPTIMIZATIONS FOR HIGH THROUGHPUT

**Decoupling DPDK Tx/Rx processing:** In the outbound direction of Onyx, we leverage DPDK to bypass the kernel and multicast market data in low latency. We adopt the high-performance lockless queue [49] provided by DPDK to decouple the Tx/Rx processing logic. On each 8-core proxy VM in Onyx, we allocate one core (i.e., one polling thread) for Rx and 6 cores for Tx (while 1 core is reserved for logging/monitoring processes). 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 messages.

**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 pre-allocated memory pools) and attach each pair of these buffers to one shallow copy created previously. Then we configure the headers (i.e., 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 inherit the sharding idea used by CloudEx [2].

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, Onyx’s throughput scales horizontally by adding more multicast trees.

## E RECEIVER HEDGING STATE SYNCHRONIZATION

### E.1 Design

In receiver hedging, we employ multiple VMs to run the same logic for one MP, so these VMs generate and submit the same orders to the exchanges. Exchanges employ an order de-duplication mechanism where every order carries a unique identifier (e.g., Order Token in Ouchv4.2, UserRefNum in Ouchv5.0 order submission protocols [50]), used by the exchange to discard duplicate orders. Thus, the identical orders generated from both receiver VMs will not be executed twice by the exchanges.

One natural concern is: what if one VM experiences multicast packet drops? This could cause the two VMs of an MP to diverge in state, potentially issuing inconsistent orders. To address this issue, we incorporate a state synchronization protocol to detect and resolve such inconsistency.

**State Synchronization:** Each order gateway, residing in receiver VMs, ensures that multicast packets are seen by the trading programs in the order of their sequence numbers attached by the exchange. Any lost packet can be identified by a missing sequence number and a retransmission (termed as *rewinding* in market data protocols [25, 51]) can be requested from a packet rewinder service. This ensures that each order generated by a VM maps to a complete prefix of multicast market data packets.

As prefixes will be identical at both receiver VMs, identical orders will be issued by the VMs of an MP as long as the trading algorithms are employed in an identical order by both VMs. As each MP may want to use multiple algorithms to process market data and generate multiple orders, we require the algorithms to be executed in an ordered sequence, identically at both VMs.

With the above mechanism, one VM’s multicast prefix may lag behind the other VM in the event of packet losses, but *a VM will never issue an order that the other VM wouldn’t or already has not issued*. A proof is presented in Appendix §E.2. Each VM calculates the unique order identifier based on the used multicast prefix, trading algorithm and the identifier of MP, and the unique order identifier will be used by exchanges to conduct order deduplication.

### E.2 Proof

For completeness, we provide a proof here.

LEMMA E.1. *Let two receiver VMs,  $R_1$  and  $R_2$ , belong to the same market participant (MP). Suppose the following conditions hold:*

- (1) *Market data messages are delivered over an unreliable channel (e.g., UDP) and are annotated with sequence numbers.*
- (2) *Each VM generates orders only upon receiving a complete prefix of market data messages up to some sequence number  $i$  (i.e., no gaps in  $[M_1, M_2, \dots, M_i]$ ).*

(3) *Each VM executes a fixed, ordered list of deterministic trading algorithms  $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_k$  over each complete prefix, and may emit exactly one order per algorithm per prefix. Orders are allowed to be empty.*

(4) *Each emitted order is tagged with a unique token computed as a hash of the market data prefix, the trading algorithm identity, and the MP ID.*

*Then, the sequence of order tokens generated by  $R_1$  and  $R_2$  is identical. One VM may lag behind the other in emitting orders, but the sequences will never diverge.*

PROOF. We prove that the sequences of emitted order tokens by  $R_1$  and  $R_2$  are identical by induction on the sequence of complete prefixes and the fixed order of trading algorithms applied to them.

Let  $P_i = [M_1, M_2, \dots, M_i]$  denote the  $i$ -th complete prefix of market data messages, and let  $\mathcal{A}_j$  be the  $j$ -th trading algorithm in the fixed ordered list. We define a deterministic order generation slot as the pair  $(P_i, \mathcal{A}_j)$ .

Let  $O_{i,j}^{(1)}$  and  $O_{i,j}^{(2)}$  denote the orders generated by  $R_1$  and  $R_2$ , respectively, for the pair  $(P_i, \mathcal{A}_j)$ . We aim to show:

$$\forall i, j, \quad O_{i,j}^{(1)} = O_{i,j}^{(2)}$$

or, if only one of the VMs has seen  $P_i$  at a given time, the other will produce the same order once it catches up i.e., the other VM also sees  $P_i$ .

We proceed by induction on  $(i, j)$  in lexicographic order:

**Base case:** For  $(i = 1, j = 1)$ , suppose  $R_1$  receives  $P_1$  first. It applies  $\mathcal{A}_1$  and emits  $O_{1,1}^{(1)}$  with a token computed as a hash of  $(P_1, \mathcal{A}_1, \text{MP ID})$ . Once  $R_2$  receives  $P_1$ , it performs the same computation deterministically and emits  $O_{1,1}^{(2)} = O_{1,1}^{(1)}$  with the same token. The sequences are aligned.

**Inductive step:** Assume for all  $(i', j') < (i, j)$ , the order tokens emitted by  $R_1$  and  $R_2$  are identical. Consider  $(i, j)$ :

(i) If both VMs have received  $P_i$ , then both apply  $\mathcal{A}_j$  to the same input using the same deterministic logic. Therefore:

$$O_{i,j}^{(1)} = O_{i,j}^{(2)}$$

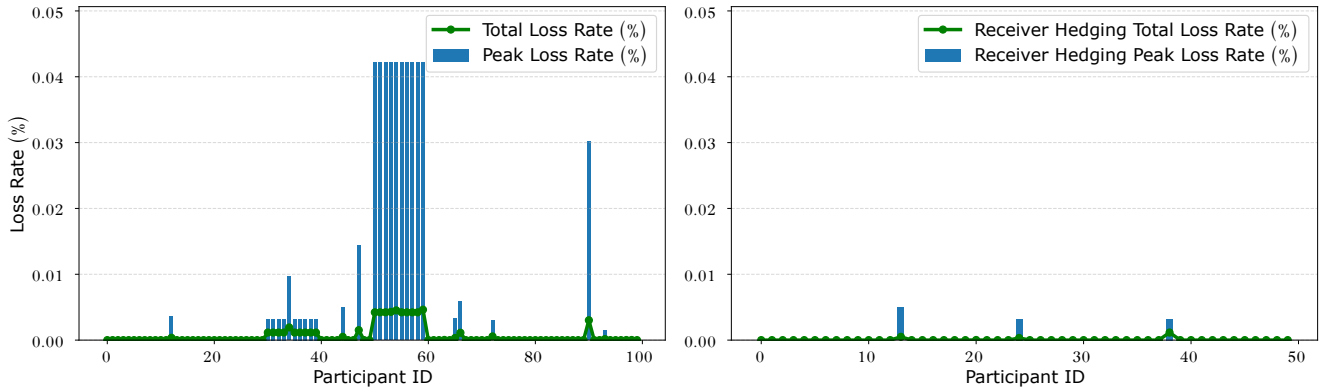
(ii) If only one VM (say  $R_1$ ) has received  $P_i$ , then only  $R_1$  is allowed to emit  $O_{i,j}^{(1)}$ ;  $R_2$  emits nothing yet. But once  $R_2$  receives  $P_i$ , it computes the same  $O_{i,j}^{(2)} = O_{i,j}^{(1)}$ . Hence, the sequence at  $R_2$  extends to match  $R_1$ ’s.

Therefore, the emitted sequences are either equal at each point, or one is a strict prefix of the other. They will eventually converge to the same full sequence.

**Conclusion:** The two VMs produce the same sequence of orders, possibly at different times, but never diverging.  $\square$

## F MULTICAST PACKET LOSSES

We employ c2d-highcpu-8 instances to run a 60-minute benchmark test (with 100 receivers) on GCP. We observe very small packet losses while multicasting at the rate of 175K messages per second (MPS). With 175K MPS, the exchange and each proxy VM sends out  $175K \cdot 10 = 1.75$  Million packets per second). Figure 18 shows that **total packet losses** over the entire duration of the experiment do not increase beyond 0.005% while the maximum losses observed



**Fig. 18: Packet Losses are low in the public cloud. Left (without any of our redundancy techniques) and right (with receiver hedging) plots show the peak loss rate as well as total losses.**

over any 1-minute duration (**peak loss rate**) are around 0.045% even without any hedging technique. Both total losses and peak loss rate decrease significantly if receiver hedging is utilized. Our results align with the statistics reported by GCP’s Global Performance Dashboard [52]. The dashboard reports 0.00262% packet-loss in the us-east4-c region for all VM-to-VM communication over a 7-day period while the peak loss rate (per minute) stayed at  $\leq 0.0174\%$ .

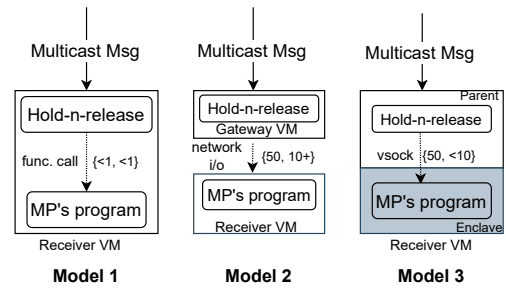
On AWS, using c5.2xlarge instances, we observe slightly higher packet losses: 0.008% lost packets with a multicast rate of 10K MPS and 1.7% lost packets with a rate of 100K MPS. For AWS, we suspect that the increased losses are due to the packet per second quotas implemented by AWS [53].

## G ONYX TRUST MODELS

In an ideal trust model, an exchange wouldn’t need to rely on MPs to adhere to protocols like accurately time-stamping trade orders or withholding market data processing until a set deadline to ensure fairness. Likewise, MPs wouldn’t have to disclose their trading algorithms to the exchange. These guarantees should be achieved without incurring any performance overhead, such as increased latency or reduced throughput.

An ideal model should not introduce any *jitter* for packets going from the hold-and-release (the exchange’s program to delay release of market data) to MPs’ trading algorithms. Fairness is achieved at the level of the hold-and-release program using CloudEx’s time synchronization mechanisms, which Onyx also leverages. To ensure fairness at the level of trading algorithms, there should be a constant latency between the exchange program and the trading algorithm. Achieving this ideal model is impractical due to the tension between security and performance that we present. We propose three trust models (Fig. 19), discuss their respective trade-offs, and offer our recommendation.

**Model 1: MPs give their programs to the exchange.** Exchange controls the VMs where the MPs’ trading programs run. The exchange runs hold-and-release mechanism in these VMs and then forwards the multicast messages to the MPs’ programs. No significant latency or throughput overhead is incurred. The jitter between the exchange’s program and the MP’s program can be minimized to



**Fig. 19: Architectures corresponding to each trust model. Dotted arrows represent messages going from an exchange’s programs to an MP’s program where the type of message is represented on the left side and  $\{a, b\}$  denotes  $a \mu s$  of latency and  $b \mu s$  of jitter for messages.**

be negligible ( $\leq 1 \mu s$ ) as forwarding messages between the two programs can be mere function calls. However, MPs have to reveal their proprietary trading programs to the exchange in this model.

**Model 2: Separate gateways for hold-and-release.** Model 2 deploys gateways between the exchange’s infrastructure (ME and proxy tree) and the VMs where MPs’ algorithms run. The exchange controls the gateways hosting the hold-and-release programs, while MPs control the VMs running their algorithms. It avoids the need for MPs to reveal their programs or for the exchange to trust MPs with hold-and-release. However, it introduces latency between the hold-and-release program and MPs’ programs, as they run in separate VMs. Throughput is unaffected, depending on VM bandwidth, but OWD between VMs is around  $50 \mu s$ , with high jitter due to cloud latency fluctuations and spikes. In the absence of the spikes, jitter may be in the tens of microseconds. Even with simultaneous delivery at the gateway level, fairness at the receiver VM level is not guaranteed due to this jitter.

**Model 3: Trading programs run in secure enclaves** Model 3 leverages the confidential computing capabilities of VMs equipped with secure enclaves such as AWS Nitro Enclave. An enclave can only talk to its associated (parent) VM (and AWS Nitro Hypervisor). The

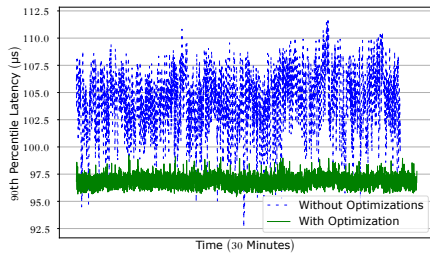


Fig. 20: RTT between a VM and its enclave stabilizes

exchange owns the VMs subjecting incoming messages to hold-and-release before forwarding them to the respective enclaves within the VMs. The MPs’ program executes within the enclave.<sup>9</sup> MPs do not have to reveal their trading programs to the exchange and the exchange does not have to rely on the MPs to run hold-and-release. Despite of the strong security boundaries, this model incurs expensive performance overhead: high latency, extremely low throughput, and non-negligible jitter. This model has been referenced in [20].

The latency between a VM and an enclave is similar to the latency between two VMs, making it comparable to Model 2. However, Model 3 benefits from reduced jitter between a VM and its enclave, likely because communication between them does not traverse the network. Optimizations described next further reduce the jitter. Nonetheless, throughput between a parent VM and its enclave is significantly lower (about 70% lower) compared to the parent VM’s ingress, as noted in [54].

**Reducing jitter between a VM and an enclave** We reduce the latency variance between a VM and its associated enclave with some optimizations: isolating CPUs, reducing scheduling-clock ticks, and pinning threads to cores. Figure 20 shows the 90th percentile latency between a parent VM and its associated enclave for each tumbling window of 1 s. After the optimizations, the latency becomes much more predictable. We observe a jitter (the difference between 90p and 50p latency) of  $\leq 10 \mu\text{s}$ .

**Recommended Model.** Optimal performance is attained with Model 1, though MPs need to reveal their programs to the exchange. If an exchange is obliged to not reveal its clients information to third parties (potentially binding due to Section 6801 of 15 U.S. Code in some territories [55]), this model should be adopted in practice. Therefore, we use Model 1 in our deployment.

## H MORE ON PROXY HEDGING

**Reduced OWDs for each receiver.** Proxy hedging improves OML as the OWD to each receiver is reduced. Figure 21 shows OWD for each receiver with ( $H = 2$ ) and without hedging.

**Associated Cost** Proxy hedging lowers multicast throughput in proxy VMs due to redundant tasks like sending messages to nieces. Reclaiming this lost throughput requires  $H$  parallel proxy trees with a shared root (sender) and leaves (receivers), assuming receivers can handle the traffic. Table 4 details the cost of proxies on AWS, based on the c5.2xlarge instance at \$0.34/hour for 100 and 1000 multicast receivers which is minuscule compared to on-premises exchange’s infrastructure cost and colocation fees [6, 56].

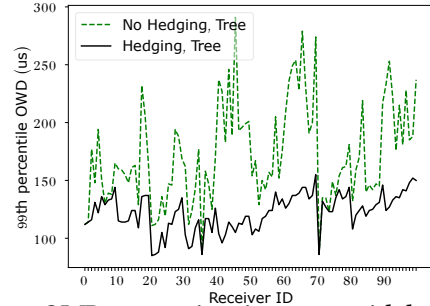


Fig. 21: OWD per receiver improves with hedging

N	# of proxy VMs (H=0)	# of proxy VMs (H=2)	Cost/hour (H=2)
100	10	30	\$10.2
1000	110	330	\$112.2

Tbl. 4: Cost of proxy hedging

## I DETAILED INTUITION OF LOQ CORRECTNESS

For any continuous sequence  $S$  of orders with the same mid-price (and thereby same value of  $I_m$ ), only the critical orders will actually be executed (i.e. matched with another order) by the ME, in the order of their timestamps. Consider a non-critical bid with value  $b < m - w$ . Since the lowest existing asking price before the sequence is executed is, by definition, greater than  $m$ , this bid cannot match with any existing ask. Then, since all asks in the sequence have value  $a \geq m - w$ , the non-critical bid cannot match with any other asks in its sequence. Non-critical orders in  $S$  will only be matched with orders that have strictly greater  $I_m$  while all orders with higher  $I_m$  have lower priority than all orders with lower  $I_m$ .

Within  $S$ , as long as the critical orders are processed in the sorted timestamp order, the resulting executions will be the same as executing *all* orders sorted by timestamp. Furthermore, if the non-critical orders are also processed in the sorted timestamp order, the state of the order book will be the same as processing each order in  $S$  sorted by timestamp. The limit order book is first sorted by value (descending for bids and ascending for asks) and then by timestamp; all the critical orders will be sorted first and then the non-critical orders will be sorted after.

The LOQ construction holds the property that the critical and non-critical orders are sorted by timestamp separately when dequeued. Then, the sequencer at each node ensures that this property is maintained when combining multiple streams consisting of LOQ outputs. Finally, since  $I_m$  is always increasing with timestamp, the sequence of orders that the ME can be partitioned into contiguous sequences of orders with the same value of  $I_m$ . Therefore, since the executions within and the state of the order book between each partition equal the executions and state if the orders were processed sorted by timestamp, the order of executions across the entire sequence or requests will be the same.

## J SEQUENCING OVERHEAD AND POTENTIAL IMPROVEMENTS

There are two types of overheads in the sequencer: (i) waiting for messages, and (ii) the overhead of packet processing within

<sup>9</sup>Loaded via remote attestation mediated by AWS Nitro Hypervisor

the sequencer. The computation within our sequencer is fairly straightforward for our packet rates; hence we focus on item (i).

The process of waiting for all downstreams' messages before releasing any message from the sequencer significantly lowers the number of messages processed per second by the exchange, making it a bottleneck. Any single trader's packet taking longer to arrive at the sequencer holds up the sequencer from processing packets of all other traders. We empirically see that the number of packets processed by the exchange decreases sharply when using a sequencer as shown by evaluation.

Below, we discuss potential ways of improving the sequencer. The first two suggestions below pertain to the overhead of waiting for messages and the third pertains to the computation overhead.

1) Running 2 (or more) instances of a sequencer where each trader submits one copy of order to each instance may improve performance. Each sequencer is one separate VM. All sequencers forward their output to the exchange VM. It is different from our current setup in the submitted paper where sequencer and the exchange may run in one VM.

More instances of a sequencer improve the chances of any one instance receiving all traders' packets early enough. However, it needs to be empirically studied whether the benefits will outweigh the new overheads i.e., (i) the exchange needs to process multiple copies of data, check them for duplication, and discard them if needed, (ii) throughput of the exchange may suffer as one extra hop is introduced in the path of packets.

2) Another improvement could be to change the protocol to not wait for messages from all clients and move on after a fixed amount of waiting period. This may improve the throughput but incur some unfairness as some participants' messages are not accounted for.

3) A sequencer also has to identify whether at least one packet has been received from each downstream VM. This check can be made faster by a better algorithm or implementing in hardware (such as within a SmartNIC if this is being eventually implemented by the cloud provider). However, this may only bring benefits when the packet rates are sufficiently high.

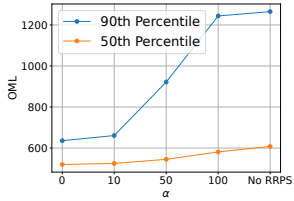


Fig. 22: IID case: benefits inversely proportional to the size of the spiky links' set.

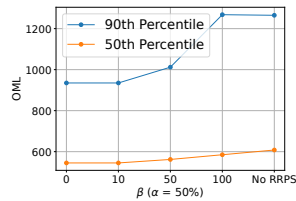


Fig. 23: Non-IID: Benefits inversely proportional to how correlated the links are.

## K RRPS MONTE CARLO ANALYSIS

To evaluate the effectiveness of RRPS under latency spikes/fluctuations, we perform a Monte Carlo analysis. The analysis shows that when a subset of links go through latency spikes (due to heterogeneity of links), packet spraying leads to

latency reduction, because messages get a chance to avoid the worse-off links.

As multiple virtual links may map to the same underlying physical links, the effectiveness of RRPS depends on how independent links are from each other. In the case where each virtual link is independent, we see the most latency reduction when the overall network performance is good, while the reduction effect diminishes as links become more and more correlated (i.e., non-IID). To capture this effect, we model the latencies as *dependent* as well as *independent* random variables.

We define OML (overall multicast latency) for a multicast message as:

$$OML = \max_{i \in \mathcal{R}} (L(S, i))$$

where:

- $\mathcal{R}$  is the set of all receivers,
- $L(S, i)$  represents latency of a multicast packet sent from the root  $S$  and received by receiver  $i$ .

**Independent Link Latencies:** In this scenario, we have all links divided into two mutually exclusive sets, each with a different link latency distribution: 1. Spiky set ( $A$ ) and 2. Non-spiky set ( $B$ ). The link latencies within each set are independent and identically distributed (*i.i.d.*). The mean of the latency distribution of the non-spiky set is 50, while that of the spiky set is 500. Each time a message goes through a link between two nodes, it uses the latency sampled from the distribution of its assigned set. Formally, let  $A = \{\ell_1^A, \dots, \ell_m^A\}$  and  $B = \{\ell_1^B, \dots, \ell_n^B\}$  be the two sets of links, let  $\mathcal{D}_A$  and  $\mathcal{D}_B$  be the two distributions with a mean of 500 and 50 each. Then, for a given time  $t$ :

$$L_{j,t}^A \sim \mathcal{D}_A, \text{ i.i.d. for } j = 1, \dots, m, \theta_A = 500$$

$$L_{i,t}^B \sim \mathcal{D}_B, \text{ i.i.d. for } i = 1, \dots, n, \theta_B = 50$$

In order to simulate the real network better, we have a latency to "stick" to that link for a few later messages as well.

Additionally, we state  $\alpha$  as the probability of a round of RRPS causing at least one proxy to send through spiky links to its receivers. For instance,  $\alpha = 0\%$  means proxies never send through spiky links, which is effectively equivalent to there are no spiky links, and  $\alpha = 100\%$  means RRPS will always make a proxy to send through spiky links. We assume that there are no spiky links from proxy to proxy.  $\alpha$  can be seen as a metric of how the overall network performs as well.

In the spike experiments, we assume that there is at least one link in-use that is spiky when the system does not use RRPS. For simplicity, we also assume that there are no spiky links from proxy to proxy. Based on Figure 22, as  $\alpha$  increases, the overall latency increases, because a message is more likely to go through a spiky link. Yet, as long as  $\alpha$  is not 100%, it outperforms not using RRPS, which constantly suffers from the spikes. The large difference in the 90th percentile further backs how RRPS leads to latency reduction by avoiding the worse-off links.

**Dependent Link Latencies:** In this scenario, we built dependencies/correlations between the two aforementioned link sets, set  $A$  with a size of  $n$  and set  $B$  with a size of  $m$ . We simulate each link's latency distribution in set  $B$  depending on that of one link in

set  $A$ . The latencies of links within each set are independent and identically distributed. Formally, for a given time  $t$ , we have:

$$\begin{aligned} L_{i,t}^A &\sim \mathcal{D}_A, \quad \text{i.i.d. for } i = 1, \dots, n, \theta_A = 500 \\ \varepsilon_{j,t} &\sim \mathcal{D}_B, \quad \text{i.i.d. for } j = 1, \dots, m, \theta_B = 50 \\ L_{j,t}^B &= g\left(L_{f(j),t}^A, \varepsilon_{j,t}\right), \quad \text{for } j = 1, \dots, m, \end{aligned}$$

where  $f : \{1, \dots, m\} \rightarrow \{\perp, 1, \dots, n\}$  maps each link in  $B$  to either a spiky link in  $A$  or  $\perp$  which represents no dependency ( $L_{\perp}^A$  is always 0), and  $g$  is a deterministic transformation function. For simplicity, we model  $g$  as a linear function. Thus we have:

$$L_{j,t}^B = \beta L_{f(j),t}^A + \varepsilon_{j,t}$$

where  $\beta$  represents how dependent  $A$  is toward  $B$ . With a similar setup as the previous scenario, we have the distribution of latencies in set  $A$  to be an exponential distribution with a mean of 500 and that of set  $B$  to be 50. For simplicity, we assume all proxy-to-proxy links are in set  $A$  (i.e., not spiky).

To produce the simulated result in Figure 23, we fix  $\alpha = 50\%$ , i.e., 50% of the time RRPS causes a proxy send through spiky links, and 50% of the time send through links that depend on spiky links. We observe that as  $\beta$  increases, e.g., as non-spiky links depend more and more on spiky links, the overall latency increases. When  $\beta = 100\%$ , the latencies suffered by the dependent links exceed those of the base links, as the former also samples from a non-spiky latency distribution. The correlation between OML and "how dependent links are" again demonstrates our claim that RRPS helps the system to avoid slow paths, and thus minimize the overall latencies.

## L COMPARISON WITH ON-PREMISES SYSTEMS

Onyx offers a different trade-off from on-premises exchanges: by relaxing some of the strict performance guarantees, it can leverage the cloud's benefits in terms of much higher scale (number of participants) and cost-effectiveness (i.e., without specialized infrastructure and carefully measured cable lengths). This in turn lowers the barrier to entry for new participants. Figure 24 shows the trade-offs curve and Onyx's position.

Additionally, for certain financial instruments (e.g., cryptocurrency), their dominant exchanges (e.g., Coinbase [57], Binance [58] etc.,) have been built on the cloud since the beginning, where such latencies are acceptable. For such instruments, Onyx is very competitive in its latency, enabling an on-prem style high frequency trading on the public cloud. We also intend to use Onyx to empirically show how far (in scale and performance) we can take a cloud-tenant centered approach to cloud-hosted exchanges. This is in contrast to recent efforts by cloud providers themselves to provide in-house support for such exchanges, often in partnership with the exchanges (e.g., [59]). While Onyx's performance will be lower than that of a system designed by a cloud provider with more intimate access to the infrastructure (e.g., the use of SmartNICs [4]), Onyx's techniques are still valuable as it shows what can be achieved immediately by cloud tenants with no help from the cloud provider.

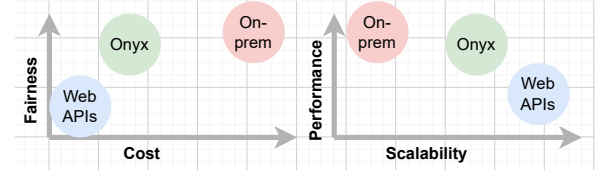


Fig. 24: Onyx Position

## M PSEUDOCODE FOR SEQUENCER

Algorithm 1 presents an efficient implementation of the sequencer's enqueue and dequeue routines. **Enqueue** is made lightweight so that incoming messages can be processed as quickly as possible and a queue formation at the ingress of the exchange's VM can be avoided. **Enqueue** is invoked at each new message while **Dequeue** runs in a separate thread indefinitely.

---

### Algorithm 1: Sequencer Enqueue and Dequeue

---

**Input:**  $n$ : Number of total downstreams (MPs)  
 $v$ : Vector of  $n$  lockless FIFO queues  
*result*: A FIFO queue for sequenced messages

```

1 Enqueue( $m_t^i$ )  $v[i].enqueue(m_t^i)$ ;
2 Dequeue()
3 while True do
4    $ts \leftarrow \infty$ ;
5    $ind \leftarrow -1$ ;
6   for  $i \leftarrow 0$  to  $n - 1$  do
7     if  $v[i].empty() = \text{True}$  then
8        $ts \leftarrow \infty$ ;
9       break;
10    end
11     $m_t^i \leftarrow v[i].top()$ ;
12    if ( $t = ts$  and  $i < ind$ ) or ( $t < ts$ ) then
13       $ts \leftarrow t$ ;
14       $ind \leftarrow i$ ;
15    end
16  end
17  if  $ts \neq \infty$  then
18     $m_{ts}^{ind} \leftarrow v[ind].dequeue()$ ;
19    if  $m_{ts}^{ind}$  is a dummy message then
20      continue; ▷ A message contains a field showing
21      whether it is a dummy.
22    end
23    result.enqueue( $m_{ts}^{ind}$ );
24  end

```

---

## N PACKET LOSSES IMPACT ON LOQ

As the exchange server multicasts market data, it can also add the mid-price ID to outgoing data. If a receiver loses a market data packet, it has stale information of mid-price until (i) it recovers the lost packet or (ii) it receives the next packet (which is not lost).

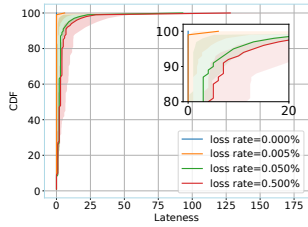


Fig. 25: Lateness proportional to losses. 10 MPs

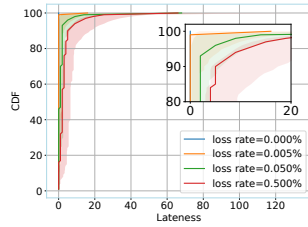


Fig. 26: Similar trend with 100 MPs.

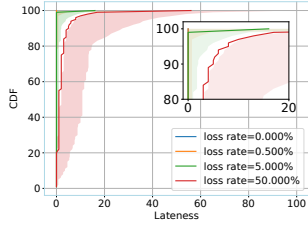


Fig. 27: 1 out of 10 experiences losses.

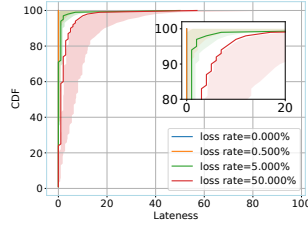


Fig. 28: 2 out of 10 experience losses.

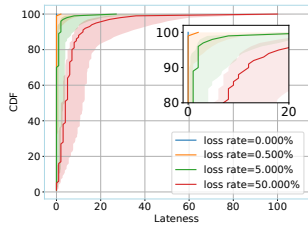


Fig. 29: 5 out of 10 experience losses.

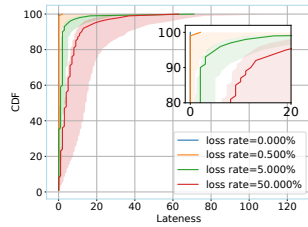


Fig. 30: 10 out of 10 experience losses.

During the period of stale mid-price, the receiver’s/MP’s orders will carry stale mid-price ID introducing non-zero lateness for orders in the exchange. We quantify this lateness as follows.

Figure 25 shows the lateness in the output sequence increases as the packet losses increase, but is sufficiently small for the typically observed losses ( $\leq 0.005\%$ ). This experiment involves 10 MPs, each sending 10K messages per second for a duration of 10 seconds. We repeat the experiment with 100 MPs, the results (Figure 26) show a similar trend.

In the above experiments, *all* MPs experience the specified packet loss rate. We perform experiments with 10 MPs, 1K orders per second rate per MP, with varying number of MPs experiencing the losses. Figures 27–30 indicate that lower the number of clients experiencing losses, lower the lateness.

## O ASSUMPTIONS SUMMARY

- (1) Clocks of participant VMs and the exchange are synchronized with negligible error (i.e., multiple orders of magnitude

better than the time resolution of interest). We achieve this using Huygen’s algorithm.

- (2) Clocks are monotonic.
- (3) Packet losses are rare. It is true for today’s public cloud infrastructure. Losses lead to brief periods of unfairness as explained in evaluation.
- (4) Order gateways running at the receiver VMs are under the control of the exchange. It holds as receiver VMs are run by the exchange, and MP’s trading programs are loaded in them.
- (5) LOQ’s fairness guarantees require simultaneous inference of mid-price at all gateways. Multicast service provides simultaneous delivery of data, hence simultaneous inference of mid-point, for vast majority of messages. Unfairness appears during rare periods of this assumption’s violation as explained in evaluation.
- (6) Gateways ensure that any generated orders by the MPs respect action window  $w$  (§5.2.1); non-compliant orders are dropped by the gateways.

## P DBO ORDER SUBMISSION RATE

Figure 31 shows Onyx achieves higher order matching rate than both CloudEx and DBO.

Comparing DBO to Onyx is not an apple-to-apple comparison. DBO ties each order submission to a received multicast message. Its fairness guarantees fall apart if an order submission uses information from any other sources e.g., second last received multicast message. While Onyx adopts fairness definitions that are used in on-premises exchanges and an order may depend on various data sources. DBO is not a generic financial exchange system as Onyx.

**Reason for DBO’s low order matching rate:** It utilizes a sequencer similar to ours but does not employ a tree or any special scheduling policy. The sequencing overhead makes it worse than both CloudEx and Onyx. Onyx compensates the overhead of sequencing via a tree and LOQ. On the other hand, CloudEx’s sequencer does not provide guarantees of inbound fairness as it only waits for a set timeout for any orders and consequently has a lower sequencing overhead (but may break fairness guarantees).

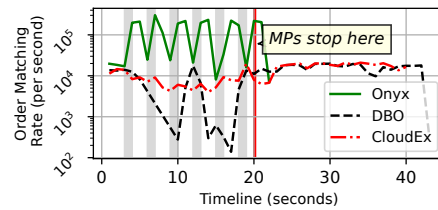


Fig. 31: Onyx achieves higher order matching rate than both DBO and CloudEx