Technische Universität München
Lehrstuhl für Kommunikationsnetze
Prof. Dr.-Ing. Wolfgang Kellerer

Master‘s Thesis

VM Selection Heuristic for Financial Exchanges in the Cloud

Author:	Daniel Duclos-Cavalcanti
Address:	230 W 55th Street
	10019 New York, NY
	U.S.A.
Matriculation Number:	03692475
Supervisors:	Navidreza Asadi,
	Muhammad Haseeb (NYU)
Begin:	03. April 2024
End:	03. October 2024

With my signature below, I assert that the work in this thesis has been composed by myself independently and no source materials or aids other than those mentioned in the thesis have been used.

München, 03.10.2024	Daniel Duclos-Cavalcanti
Place, Date	Signature

München, 03.10.2024	Daniel Duclos-Cavalcanti
Place, Date	Signature

Kurzfassung

Cloud-Computing hat die Bereitstellung von Softwarediensten revolutioniert, indem es virtualisierte Rechenressourcen bedarfsgerecht zur Verfügung stellt. Finanzbörsen untersuchen das Potenzial einer Migration in die Cloud, um eine verbesserte Skalierbarkeit und geringere Betriebskosten zu erreichen. Die inhärente Schwankung der Netzwerkleistung und das Fehlen nativer Multicast-Lösungen in der Cloud stellen jedoch erhebliche Herausforderungen dar, um eine faire und latenzarme Datenübertragung zu gewährleisten, was für moderne Börsen von entscheidender Bedeutung ist.

Jasper ist ein wegweisendes Projekt, das eine Multicast-Lösung für cloudbasierte Finanzbörsen implementiert. Es erreicht eine extrem niedrige Latenz durch eine Overlay-Proxy-Baumstruktur, Kernel-Bypassing, Uhrensynchronisation und weitere Techniken. Jasper geht jedoch nicht ausreichend auf das Problem der *"Straggler"* ein – leistungsschwache virtuelle Maschinen (VMs) innerhalb eines Clusters –, die die Systemleistung erheblich beeinträchtigen können.

Diese Arbeit stellt TreeBuilder vor, ein Werkzeug, das Jaspers Ansatz durch eine intelligente Auswahl von Knoten in einem Cluster zur Bildung eines Multicast-Baums verbessert, wobei speziell die Präsenz von Straggler-VMs berücksichtigt wird. TreeBuilder verwendet einfache Heuristiken zur Verbesserung der Latenzleistung und bietet dabei eine flexible und modulare Architektur, die schrittweise Aktualisierungen und eine einfache Anpassung an andere Netzwerktopologien ermöglicht. Wir implementieren mehrere Auswahlstrategien, darunter `WEIGHTED`, `P90` und `P50`, um ihre Auswirkungen auf die Leistung eines Multicast-Baums zu bewerten.

Durch umfangreiche Experimente zeigen wir, dass TreeBuilder vergleichbare Leistungen wie LemonDrop, eine jüngere und anspruchsvolle Heuristik zur Auswahl und Planung von VMs, erbringt, während es eine leichte und effektive Technik mit erheblichem Skalierungspotenzial beibehält. Eine Instanz des `BEST-WEIGHTED`-Baums von TreeBuilder wies die niedrigste 90. Perzentil-Latenz für ihren schlechtesten Empfänger von 312 µs auf, was etwa 7,7 % niedriger ist als die des schlechtesten Empfängers des LemonDrop-Baums (`LEMON`), der eine Latenz von 338 µs hatte. Ebenso betrug die mittlere Latenz des schlechtesten Empfängers des `BEST-WEIGHTED`-Baums 223,1 µs, was etwa 10,6 % niedriger ist als die des `LEMON`-Baums mit 249,5 µs. Allerdings wies der LemonDrop-Baum im Laufe des Experiments niedrigere Standardabweichungswerte auf, was auf ein stabileres System aufgrund des Median-basierten Optimierungsansatzes hinweist. Zusammenfassend bietet TreeBuilder eine zugängliche, erweiterbare Lösung für die Multicast-Übertragung in Cloud-Umgebungen mit erheblichem Potenzial für weitere Optimierungen und Anwendungen in breiteren verteilten Architekturen.

Abstract

Cloud computing has revolutionized the deployment of software services by providing virtualized computational resources rented on-demand. Financial exchanges are exploring the potential of migrating to the cloud for improved scalability and reduced costs. However, the cloud's inherent variance in network performance and lack of native multicast solutions present challenges to achieving fair and low-latency data dissemination, which are essential for modern exchanges.

Jasper is a pioneering project that implements a multicast solution for cloud-based financial exchanges. It achieves ultra-low latency through an overlay proxy tree structure, kernel bypassing, clock synchronization, and more. However, Jasper does not fully address the issue of *"stragglers"* – underperforming virtual machines (VMs) within a cluster – which can significantly degrade system performance.

This work introduces TreeBuilder, a tool designed to enhance Jasper's approach by intelligently selecting nodes in a pool to form a multicast tree, specifically addressing the presence of straggler VMs. TreeBuilder employs straightforward heuristics to improve latency performance while maintaining a flexible and modular architecture that allows for incremental updates and easy adaptation to other network topologies. We implement several selection strategies, including `WEIGHTED`, `P90`, and `P50`, to evaluate their impact on a multicast tree's performance.

Through extensive experiments, we demonstrate that TreeBuilder performs comparably to LemonDrop, a recent and sophisticated VM selection and scheduling heuristic, while maintaining a lightweight, effective technique with considerable scalability. An instance of TreeBuilder's `BEST-WEIGHTED` tree exhibited the lowest 90th percentile latency for its worst receiver at 312 µs, approximately 7.7% lower than the worst receiver of LemonDrop's tree (`LEMON`), which had a latency of 338 µs. Similarly, the `BEST-WEIGHTED` tree's worst receiver's mean latency was 223.1 µs, around 10.6% lower than that of `LEMON` at 249.5 µs. However, LemonDrop's tree exhibited lower standard deviation values throughout the experiment, indicating a more stable system due to its median-based optimization approach. In conclusion, TreeBuilder offers an accessible, extensible solution for multicast dissemination in cloud environments, with significant potential for further optimization and application to broader distributed architectures.

Contents

Chapter 1

Introduction

Cloud computing has transformed completely the landscape of software and service deployment. By providing Infrastructure as a Service (**IaaS**), cloud providers have allowed users to conveniently access scalable, robust and virtualized computing resources on a demand-driven basis. What once required considerable investments in hardware and technical expertise is now accessible instantaneously, significantly lowering the barrier of entry to sophisticated networked infrastructure.

Financial exchanges aim to leverage this medium for several reasons such as improved scalability and reduced operational costs. Despite the clear benefits brought by the cloud, there are still issues inhibiting its adoption by modern exchanges. These have traditionally operated in on-premise data-centers, highly-engineered to ensure deterministic and predictable low-latency conditions. This is crucial to enforce fairness in the market, where participants receive market updates simultaneously. To achieve this, L1 switches are employed to perform low-latency physical-layer multicast [lep24] and FPGAs are frequently used as gateways [nov24]. In addition to that, exchanges go as far as equalizing cable lengths between themselves and participants servers, as to eliminate all possible sources of latency deviations. However, the cloud lacks native solutions for such enhancements and exhibits inherent variance in network performance and resource allocation. Thus, posing a significant challenge towards the migration of financial exchanges to the public cloud.

A notable project in this domain and the first of its kind is Jasper [HGB+24]. This work designs and implements a multicast service for cloud-hosted exchanges, achieving ultra-low latency while scaling to a large number of receivers. Jasper introduces techniques, such as overlay proxy trees, kernel by-passing and clock synchronization across nodes, to achieve fair and performant market data dissemination. Nonetheless, there is still opportunity for enhancement, as linked to the clouds volatile nature is the occurrence of *"stragglers"* in a cluster. Stragglers are Virtual Machines (**VMs**) that considerably underperform relative to others despite having identical configurations. These can occur due to the sharing of computational resources across VMs [XMNB13] and can negatively impact latency-sensitive applications such as that of a financial exchange. Efforts have been made to predict and manage lesser performing VMs. Among these, LemonDrop [Sac22] presents a fitting approach, directly considering latency measurements between machines to optimize for an application's network performance. However, this work is disproportionally robust, incurring significant computational overhead and offering scalability issues considering a typical financial exchange scenario.

This thesis introduces TreeBuilder, a tool specifically designed to intelligently select nodes in a cluster to form a multicast tree. TreeBuilder builds upon Jasper's overlay proxy tree structure, aiming to further enhance its reliability and performance of data dissemination by mitigating the impact of straggler VMs in cloud environments. In addition to employing straightforward heuristics that achieve considerable latency improvements, TreeBuilder features a flexible and modular architecture, making it easily extensible and adaptable for various network topologies.

The rest of this work is organized as follows. Chapter 2 presents the background to our project and relevant research work within its scope. Chapter 3 describes TreeBuilder's proposed method, goals and high-level design. In Chapter 4, we detail its implementation, whereas in Chapter 5, we discuss the experiments and their results relative to our design goals. Finally, we conclude the thesis in Chapter 6.

Chapter 2

Background

In this chapter, we discuss the required background on which this work builds. We start by discussing a quick overview of cloud computing and modern straggler optimization techniques. Finally, we briefly introduce financial exchanges, the challenges of their migration to the cloud and current efforts in the matter.

2.1 Cloud Computing

Before personal computers became widespread, they were famously expensive for the average consumer. It wasn't until the 1980s and 1990s that this changed. However, building large-scale applications still required significant knowledge in configuring, managing, and maintaining extensive computing resources. Such complexity often limited the development of software applications to well-funded organizations with dedicated IT expertise. However, in 2006 Amazon introduced the Elastic Compute Cloud (EC2) [Ser06], a service that revolutionized this landscape by abstracting away the complexities of infrastructure management.

EC2 offers units of compute, memory, and storage, all connected through a networking infrastructure that enables communication between instances and, through advancements in hardware virtualization, rents units of computation per time through **VMs**. This brought forth a paradigm well known today as cloud computing, allowing for a more flexible use of compute, as machines could be up- and down-scaled more efficiently and machine maintenance could largely be outsourced. Many corporations such as Google, IBM, and Microsoft have followed, and today the public cloud is an established medium for application deployment [Clo08, IBM11, Mic10].

Cloud computing facilitated access to computing resources, reducing the barrier of entry to smaller businesses and providing a scalable and cost-effective platform for deployment. Moreover, it is a significant technological industry, generating global spending that exceeds $700 billion per year [Mic21].

2.1.1 Communication Latency

Even with the several benefits that the cloud provides, there are still inherent trade-offs in its utilization. Public clouds exhibit high latency variance, with spikes reaching 2,900x the average latency [HCW+24]. This can occur due to what is known as *"noisy neighbor"* syndrome, where one tenant's resource usage temporarily degrades the performance of others. Given the innate sharing of resources in the context of virtualization, unpredictable latency conditions can break guarantees of consistency needed by low-latency applications. Furthermore, cloud providers dynamically manage their networking fabric to optimize performance and handle varying traffic loads, consequently changing the network paths a user's data takes over time. Finally, even when requesting VMs in a cluster to be placed in a given availability zone, the exact placement of these instances in a provider's physical infrastructure can vary. All of these factors contribute to the volatile networking conditions in the cloud.

2.1.2 Stragglers in the Cloud

In addition to the cloud's variable nature, another challenge in cloud environments is the existence of *"stragglers"* – underperforming VMs in any given cluster. These are instances that, despite an identical configuration to other nodes, perform significantly worse. Research has shown that stragglers can occur due to the sharing of CPU and memory resources across VMs [XMNB13]. Stragglers pose a challenge in maintaining predictable and efficient cloud-based computations, significantly delaying the overall completion of tasks. This has critical effects, particularly in environments where large-scale distributed computing takes place, reducing system throughput, causing inefficiencies, and ultimately leading to higher operational costs.

Research efforts have leveraged machine learning to predict and avoid stragglers, optimizing resource allocation, and intelligently selecting VM configurations to enhance overall job performance in a cluster [YAK14, YHG+17, YHGK15]. These optimizations have been especially critical in the context of batch data processing frameworks, where task completion time can heavily depend on the performance of the slowest VM within a group. As cloud infrastructure continues to evolve, ongoing research aims to further enhance the detection and handling of stragglers, ensuring that applications deployed in the cloud remain resilient and performant, even when encountering unpredictable resource contention.

2.1.3 LemonDrop

One notable advancement in the field of optimizing straggler VMs is LemonDrop [Sac22]. Prior work such as [YAK14, YHG+17, YHGK15] took a rather singular focus on computationally underperforming machines, as opposed to additionally considering latency implications in their methodology. LemonDrop distinguishes itself in its approach as it uses accurate one-way delay (**OWD**) measurements between VMs and an optimization framework to select and schedule VMs directly based on an application's networking needs.

LemonDrop is particularly tailored to latency-sensitive applications, making it effective in interactive cloud environments and valuable for autoscaling and maintaining responsiveness. To achieve these objectives, LemonDrop makes use of a series of techniques.

Clock Synchronization

LemonDrop utilizes an accurate clock synchronization algorithm, allowing it to detect fine-grained anomalies in OWD between pairs of VMs [GLY+18]. This approach allows LemonDrop to uncover subtle performance variations that could otherwise go unnoticed and is crucial to obtaining quality measurements used by its selection heuristic.

Optimization Framework

LemonDrop frames the selection and scheduling process as a natural Quadratic Assignment Problem (**QAP**) [KB57]. This allows for an efficient assignment and scheduling of K VMs from a pool of $N > K$ machines. LemonDrop's optimization routine leverages two sets of data to maximize an applications network performance:

- ***Δ: OWD Matrix***. Given N VMs, LemonDrop first runs an all-to-all probing mechanism, where each pair of VMs exchange timestamped 44-byte UDP packets. This creates the $\Delta \in R^{N \times N}$ OWD matrix that captures the bi-directional latencies between all VM pairs in the system, where each entry δ_{ij} represents the median OWD latency from VM i to VM j.
- ***Λ: Application Load Matrix***. The $\Lambda \in R^{L \times L}$ load matrix captures the communication patterns of an application. Specifically, for each node pair k and l, λ_{kl} represents the rate of requests from application node k to application node l. The load matrix is either gathered offline using trace collection tools or instrumented using logging utilities. The load information provides additional insights into which nodes are expected to communicate more frequently and impacts the routine's VM selection and scheduling decisions.

Using the Λ and Δ matrices, LemonDrop formulates a QAP whose objective is to find the optimal mapping of service nodes to VMs in a given cluster. The sum of entries of the Hadamard product of Λ and Δ captures the total time requests spend traveling between VMs. This is shown below in equation 2.1.

$$D = \sum_{i,j=1}^{L} \lambda_{ij} d_{ij} \tag{2.1}$$

This operation reflects the *"in-network time"* of requests given a specific assignment of nodes to VMs, where node i is assigned to VM i. The VM scheduling problem aims to minimize this Hadamard product, by finding the best 1-1 assignment of nodes onto VMs.

Therefore, minimizing a system's in-network time of requests made. Consider a general assignment of service nodes to VMs, represented by an $L \times L$ permutation matrix P. In this matrix, $P_{ij} = 1$ indicates that service node i is assigned to VM j, while all other entries in row i and column j are zero. The "in-network time" associated with this assignment is then given by:

$$D(P) = \text{trace}(\Lambda P \Delta^T P^T) \tag{2.2}$$
$$\text{subject to} \quad P \in \mathcal{P}, \tag{2.3}$$

Finally, the entire VM scheduling problem can be summarized as:

$$\text{minimize trace}(\Lambda P \Delta^T P^T) \tag{2.4}$$

QAP is known to be NP-hard [KB57]. To overcome this, LemonDrop employs an efficient heuristic known as the Fast Approximate Quadratic Programming (**FAQ**) algorithm [VCL+14]. This technique relaxes the problem to allow for a doubly stochastic assignment matrix and then applies the Frank-Wolfe optimization method to obtain a locally optimal solution, as seen in algorithm 1. The final output is a permutation matrix P, which is projected onto the set of valid permutations to obtain the best assignment of VMs. This gives us the mapping of application nodes to VMs in a cluster and enables LemonDrop to identify an optimal subset of VMs, even under time constraints.

Algorithm 1 FAQ Algorithm [VCL+14]

```
1: function FAQ($\Lambda$, $\Delta$, $\epsilon$, $n$)
2:     $P^{(0)} = 11^T$
3:     $i = 0$
4:     $s = 0$
5:     while $s = 0$ do
6:         $\nabla f(P^{(i)}) = -\Lambda P^{(i)} \Delta^T - \Lambda^T P^{(i)} \Delta$    ▷ Gradient wrt current solution
7:         $Q^{(i)} = \arg\min_{P \in \mathcal{D}} \text{trace}(\nabla f(P^{(i)})^T P)$    ▷ Direction which minimizes 1st order
   approx of $f(P)$
8:         $\alpha^i = \min_{\alpha \in [0,1]} f(P^{(i)} + \alpha Q^{(i)})$    ▷ Best step size in chosen direction
9:         $P^{(i+1)} = P^{(i)} + \alpha^i Q^{(i)}$    ▷ Update our current solution
10:        if $\|P^{(i)} - P^{(i-1)}\|_F < \epsilon$ then $s = 1$    ▷ Stop opt
11:        if $i > n$ then $s = 1$    ▷ Stop opt
12:        $i = i + 1$    ▷ Iteration number
13:    end while
14:    $P_{\text{final}} = \arg\min_{P \in \mathcal{P}} P^{(i-1)} P^T$
15:    return $P_{\text{final}}$
16: end function
```

2.2 Financial Exchanges

Financial exchanges are a natural evolution of the traditional marketplace concept. By providing a centralized platform to exchange goods, they facilitate the continuous interaction between buyers and sellers. Market Participants (**MPs**) can engage in bidding, which involves offering to purchase an asset at a specified price, and asking, which refers to the intention to sell an asset at a specified price. This process, known as price discovery, plays a crucial role in determining the fair market value of any asset. A true and fair price benefits both buyers and sellers, since a seller wishes to obtain just compensation, while a buyer wishes to pay nothing beyond what is needed.

Modern financial exchanges achieve this by utilizing highly-engineered infrastructures, consisting of on-premise data-centers designed to ensure a fair and efficient digital marketplace. These facilities are optimized for low-latency communication, enabling participants to execute trades quickly and reliably, while ensuring that all market participants, co-located within the data center, have equal access to market information and trading opportunities. The key components of a financial exchange are outlined in the following sections and are also illustrated in Figure 2.1.

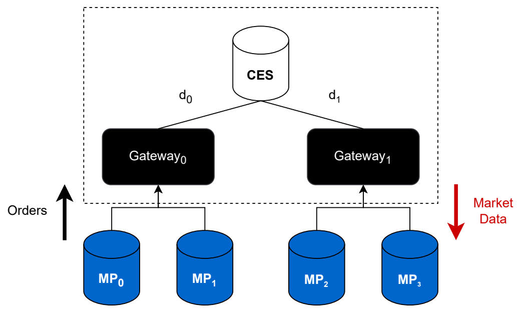


Figure 2.1: On-premise data-center diagram

Orders. Trading orders can be either bids or asks. As mentioned previously, bid orders represent intentions to buy an asset at a specified price, while ask orders indicate an offer to sell an asset at a specified price. Another important concept is the Limit Order Book (**LOB**), which is an aggregation of all bids and asks for a specific asset, providing a comprehensive summary of the asset's current market valuation.

Central Exchange Server. At the core of the Central Exchange Server (**CES**) lies a Matching Engine (**ME**) that receives and processes trading orders from market participants. Orders are sent to the CES, which in return may or may not match an incoming order with a suitable counterpart. Arriving orders, matched orders, as well as periodic snapshots of the market (LOBs) are events that are broadcasted from the CES to MPs.

Market Participants. Market participants are the co-located traders that are directly connected to an exchange's system, issuing orders and receiving data from the exchange.

Gateways. The gateways are structures placed between traders and the CES. Their responsibilities consist of routing orders and market data, as well as protecting the CES from abuse, e.g., unauthenticated or invalid orders. Within the on-premise clusters, gateways are made to be equidistant to the CES to ensure fairness regarding communication delays between themselves and the central exchange.

2.2.1 Cloud Migration

Given the demanding network performance requirements of financial exchanges, on-premise data centers remain the prevailing standard for ensuring both fair and efficient data dissemination. A fundamental characteristic of a financial exchange is the ability to broadcast market updates to market participants (MPs) in a fair manner – meaning that all MPs receive market data almost simultaneously to prevent unwanted arbitrage opportunities.

Despite the well-known advantages of public clouds, such as flexibility, scalability, and robustness, they also exhibit significant latency variance [HCW+24] and lack the low-level engineering control available to cloud tenants, such as switch-based multicast. As a result, cloud-based systems often implement multicasting by directly unicasting a copy of a message to each recipient [GSPR22, GGS+21, GLR+24, GGM+23]. This method does not scale and incurs significant latency differences across receivers as the number of recipients increase. Although, exchanges desire to move their infrastructure to the cloud, without native solutions to enable performant multicasting, a clear problem is presented.

2.2.2 Jasper: A Scalable Financial Exchange in the Cloud

A significant project in this domain is Jasper [HGB+24], which offers a modern implementation of a financial exchange on the public cloud. This work realizes a multicast service that achieves low latency of less than 250 µs and a latency difference of less than 1 µs across 1,000 multicast receivers. To accomplish this, a series of techniques are employed.

Overlay Proxy Tree. This is the starting point for Jasper's design and borrows the concept of trees to scale communication to a larger number of receivers, while improving latency in comparison to direct unicasting [CEM07, SST09]. The structure of the tree with Depth (**D**) and fan-out (**F**) has been empirically established, as seen in Equations [2.5, 2.6].

$$F = 10 \tag{2.5}$$
$$D = \log_{10}(N) \tag{2.6}$$

Clock Synchronization. Jasper utilizes high-precision software clock synchronization to compensate for noisy conditions in the cloud [GLY+18]. Additionally, a common clock across a cluster allows for precise global timestamps to coordinate actions across nodes.

Hold-and-Release. Even with a proxy tree, there are still serial delays that impact every hop in the system. Therefore, receivers located at the leaves of this tree would certainly not receive packets simultaneously. By leveraging global timestamps, a technique called hold-and-release is applied, where a chosen deadline is appended to every packet, instructing receivers when they are allowed to consume the sent data. In this way, a temporal barrier is used to enforce simultaneous updates from the market to all recipients. The deadline is set to a future point in time at which all receivers are highly likely to have received the message. To calculate these deadlines, the sender or root of the tree continuously gathers one-way-delay measurements between itself and all receivers in the tree. Cloud network variability can impact Jasper's deadline mechanism, which relies on high-percentile OWD values from its leaf nodes. This directly influences delivery windows, affecting the system's ability to maintain consistent and predictable low-latency.

VM Hedging. This optimization technique was developed to partially overcome the cloud's inherent variant nature. Due to dynamic routing and unpredictable spikes in latency, Jasper utilizes the concept of hedging, in which each node in the proxy tree receives message copies from different sources and only processes the earliest received message copy, forwarding it, and only it, to child nodes. In this way, bad links or pathways that are affected by a fluctuation in the cloud are countered by hedging message delivery on multiple pathways. When successfully optimizing against latency spikes, the overall multicast OWD latency of messages is reduced, allowing Jasper to generate tighter deadlines in future messages and, therefore, improving the overall system's fair delivery of data.

Finally, Jasper's combination of an overlay proxy tree, high-precision clock synchronization, a hold-and-release mechanism, and VM hedging enables a highly scalable and fair multicast system suitable for financial exchanges in the cloud. By addressing both the scalability requirements and fairness demands critical to financial systems, Jasper presents a compelling solution that outperforms existing cloud-based multicast approaches.

Chapter 3

Proposed Method

In this chapter we present the primary goals and design constructs of our developed distributed tool named *TreeBuilder*. TreeBuilder leverages a straight-forward heuristic to filter out stragglers from a deployed cluster and form an overlay multicast proxy tree of a given depth and fan-out. Provided that Jasper leverages the same structure to multicast market data [HGB+24], TreeBuilder potentially improves the performance of Jasper's system by decreasing overall multicast latency as it removes undesired VMs links from the tree. Additionally, we provide a truthful implementation of LemonDrop's adapted FAQ algorithm [VCL+14] within TreeBuilder, employing it as a valuable benchmark against our tool.

3.1 Goals

Goal 1: Minimizing Straggler Effect. Our primary and most important goal is to optimize against stragglers in a cluster. Under-performing VMs can heavily impact a system's performance, especially in a tree-like topology where parent nodes directly dictate the delay latency of it's children and further descendant layers.

Goal 2: Lightweight and Scalable. Our second goal, as to differentiate ourselves and to better adapt to the network requirements of modern financial exchanges, is to provide a lightweight solution. That means reduced bandwidth use, faster convergence and a scalable design.

Goal 3: Incremental Updates. The third goal, which ties well together with Goal 2, is to provide incremental updates as a feature as well. Given a tree topology, there are evident opportunities to apply the tool onto sub-trees of itself and therefore expose a natural and recursive partial update mechanism within TreeBuilder. That way, the user has the option to re-structure subsets of the system, synergizing well with the lightweight nature of this work.

3.2 Design: TreeBuilder

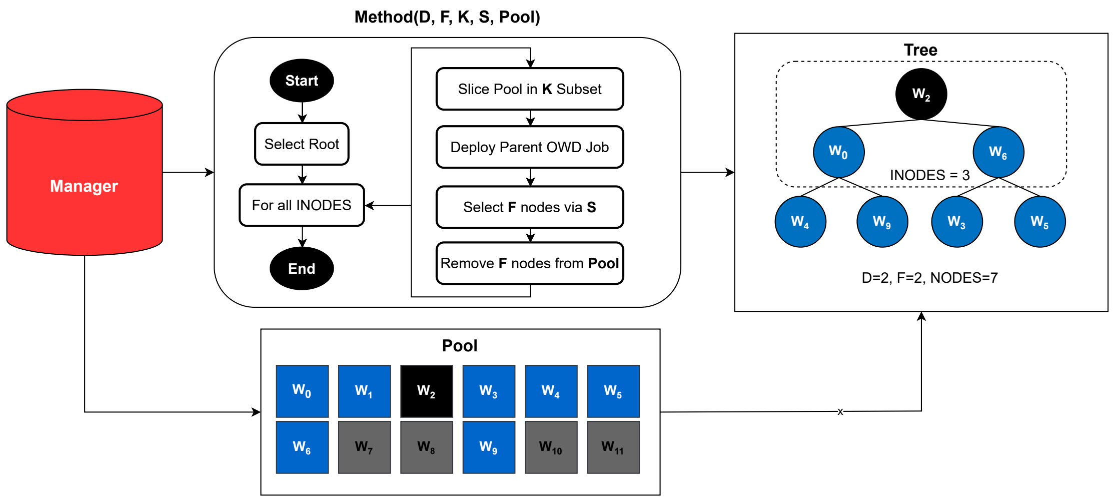


Figure 3.1: TreeBuilder's System Design Overview

TreeBuilder is designed to intelligently select a subset of nodes in a cluster to form an optimal multicast tree of a predefined structure, as illustrated in Figure 3.1. Its main objective is to mitigate the impact of stragglers and do so in a lightweight and scalable fashion, allowing for both incremental and total updates to the system. Below are the high-level parameters that govern TreeBuilder's proposed method:

- **D**: Target depth of multicast tree.
- **F**: Fan-out factor of multicast tree.
- **P**: Allocated pool of VMs in the cluster.
- **K**: Hyperparameter that bounds the size the examined node subsets.
- **S**: Strategy to select VMs from a given subset.

Given these parameters, TreeBuilder's approach can be summarized as launching a pool of nodes **P**, and through a gradual procedure of examining subsets of the pool bounded by a hyperparameter **K**, select nodes via a strategy **S** to ultimately form a multicast tree of depth **D** and fan-out **F**.

The tool's design can be modularized into three main segments: (i) Node Communication, which refers to how data and messages are exchanged within the system, (ii) Node Subset Examination, the process of evaluating node subsets based on performance metrics, and (iii) VM Selection Method, the overall procedure for selecting optimal virtual machines to form a multicast tree. The following sections will explain these concepts in more detail and outline the terminology used throughout this work.

3.2.1 Node Communication

The notion of independent nodes is fundamental to the operation of distributed applications. In this work we treat the software logic running within any given VM in our cluster as a node. Nodes in our system exchange messages under a custom application level protocol to issue commands, gather data and report on a given node's state.

Manager x Workers

Within TreeBuilder's deployed cluster, there are two types of nodes: a single primary node known as the *manager* and a set of *workers*. The workers are logically grouped into a virtual pool managed by the manager. The manager distributes *jobs* to subsets of workers, who execute the related tasks and return their results. This design enables work to be distributed across multiple worker groups while results are aggregated by the manager. This simplifies worker logic, reduces faults, and enhances predictability and scalability (**Goal 2**). Additionally, since results must ultimately be processed by the manager, this centralized approach improves the transparency of event logging, both for data storage and for retracing sequence of events.

3.2.2 Node Subset Examination

A core aspect of any heuristic aimed at selecting or scheduling VMs in a cluster is the ability to examine a set of machines and infer objective metrics regarding their performance. In our system, it is essential to perform such examinations on subsets of the node pool in an incremental and computationally efficient manner, in order to filter out stragglers while remaining lightweight and performant. The hyperparameter **K** serves as the limiting number of workers from a given subset to be evaluated at any given moment (**Goal 2**).

Clock Synchronization

As in the Jasper and LemonDrop approaches [HGB+24, Sac22], TreeBuilder chooses to leverage high-precision software clock synchronization in its design [GLY+18]. Precise global timestamps are essential to both coordinate actions across nodes, as well as to measure one-way delays between VMs or VM paths.

Subset Examinations: Jobs

Jobs refer to the tasks performed by a subset of workers. These tasks are coordinated and executed as a distributed application to evaluate and assess the performance of specific VMs in a given arrangement. For every job, there is always one worker node responsible for synchronizing its fellow worker nodes' actions and ultimately gathering their generated results. This node is referenced in this work as the job *leader*. The intention of assigning leaders is to reduce the communication burden on the manager, which would otherwise have to periodically query all nodes assigned to a job's given subset (**Goal 2**). For every job, all worker nodes in the specified group spawn a separate process running a predefined executable related to the task at hand and their role in it. When the process finishes execution, the worker node reads, parses, and stores the data streamed to standard output in running memory. The leader will probe the remaining workers at defined intervals until all results are collected. The manager also continuously probes the leader, eventually retrieving the produced results. Probing is an established mechanism in distributed systems to ensure timely synchronization of results or actions across nodes. While polling or probing is often considered more basic compared to asynchronous techniques, its simplicity offers significant advantages in terms of reliability and ease of implementation. Probing provides a clear-cut process for updates on state, making it easier to debug and monitor the system. Moreover, it aligns well with our plan of making worker nodes as simple as possible to allow for better scalability and predictability (**Goal 2**). Additionally, probing for results is subject to a timeout limit proportional to the job's duration. If this timeout is breached, TreeBuilder assumes the job has failed and will gracefully terminate the experiment, flushing the complete logs into a file. TreeBuilder defines three types of jobs:

- ***Parent Jobs:*** This task involves a subset of workers and a selection strategy **S**. One of these workers is pre-selected as the *parent*, while the remaining workers act as potential children. The purpose of this job is to obtain OWD measurements between the parent and its potential children, providing insights into which workers are the best fit to be direct descendants in the multicast tree (**Goal 1**). The parent sends timestamped packets to the children, who calculate the OWD based on the arrival times of these packets. The selection strategy is applied once the manager gathers results from each child node associated with this job. The results of a parent job are stored in a dictionary-like data structure, containing statistical metrics such as 90th percentile latency, median latency, and the standard deviation of OWD latencies as seen in Appendix A.4.2. Depending on the strategy, the manager may rank the children based on a key metric or a weighted combination of metrics. This job is used iteratively to construct the multicast tree.

- **Mcast Jobs:** Once a multicast tree is formed, this job evaluates its performance. The root of the tree, also the job leader, sends timestamped packets to its children, who relay them to their own children, and so on, until the packets reach the leaves. The leaves calculate the OWD measurements from the root to themselves upon arrival. After the job completes, the root initiates an upward traversal of the tree, collecting results in an all-reduce fashion as each child recursively gathers results from its own children.

- **Lemon Jobs:** This job implements the data-gathering procedure as outlined in LemonDrop's work [Sac22]. In this task, every node in the pool sends packets to every other node in an all-to-all probe mesh. Each worker calculates the median OWD latency between itself and every other node, which the manager gathers to construct the **OWD Matrix (Δ)** (2.1.3) used by LemonDrop's FAQ routine [VCL+14].

3.2.3 VM Selection Method and Evaluation

The method by which TreeBuilder filters out unwanted nodes and forms its multicast tree is central to this work. As previously introduced, TreeBuilder operates on a pool of nodes **P**, gradually selecting workers through a strategy **S** and bounding the examined subset by the hyperparameter **K**. This process forms a multicast tree of depth **D** and fan-out **F**. The main steps in this selection routine can be separated as (i) root selection and (ii) children selection, which is then followed by a (iii) tree evaluation procedure .

Algorithm 2 TreeBuilder Selection Method

```
1: function BuildTree(D, F, K, S, Pool)
2:     $N = \frac{F^{D+1}-1}{F-1}$                          ▷ # of nodes (A.1)
3:     $I = \frac{F^{D+1}-1}{F-1} - F^D$                    ▷ # of inodes (A.4)
4:     $L = F^D$                                            ▷ # of leaves (A.3)
5:     $root \leftarrow$ select_root($Pool$)         ▷ randomly selects root from the pool (1)
6:     $tree \leftarrow$ Tree($D, F, I, L, root$)    ▷ initialize to-be-returned tree data structure
7:     while $i < I$ do                                 ▷ Children selection (2)
8:         $parent \leftarrow tree$.next()                  ▷ Get next internal node
9:         $children$[] $\leftarrow$ slice_pool($Pool, K$)   ▷ Slice pool into a subset of size K (2a)
10:        $results \leftarrow$ deploy($parent, children$)   ▷ Deploy Parent x Children Job (2b)
11:        $chosen$[] $\leftarrow$ select($S, F$)            ▷ Select F nodes via strategy S (2c)
12:        $tree$.append($parent, chosen$)                   ▷ Add chosen VMs to the tree
13:        remove($Pool, chosen$)                            ▷ Remove F nodes from pool P (2d)
14:        $i = i + 1$
15:    end while
16:    return $tree$
17: end function
```

Root Selection

This step (**1**) consists of the random selection of a root node. This randomness is a simple, initial strategy that can be further refined in future iterations. Furthermore, it provides a baseline for comparison with more advanced strategies and minimizes the risk of biased selections at the start of the process.

Children Selection

Having established the tree root, TreeBuilder enters an iterative step (**2**) to build the multicast tree from the root downwards. Starting with the root as the selected parent, but done so for every internal node in the to-be-formed tree, TreeBuilder randomly selects **K** workers from the pool (**2a**) to be examined as the parent's potential children. Now, the corresponding parent job is then deployed (**2b**) onto this group of workers and its results are gathered by the manager. The manager ranks the produced results per node via the chosen strategy **S**, selecting **F** nodes (**2c**), and as a result removing them from the pool **P** (**2d**). Additionally, the ability to re-select nodes through the re-launch of parent jobs inherently allows for partial updates to the tree. This is especially useful for incremental improvements to the system without requiring a complete recreation of the tree (**Goal 3**).

Tree Evaluation

Once the multicast tree is formed, a mcast job is deployed onto its nodes to evaluate its performance (**3**). As previously mentioned, the tree's leaves calculate the OWD measurements from the root to themselves over the task's duration. After completion, the root collects results in an all-reduce fashion, which are then subsequently retrieved by the manager.

Chapter 4

Implementation

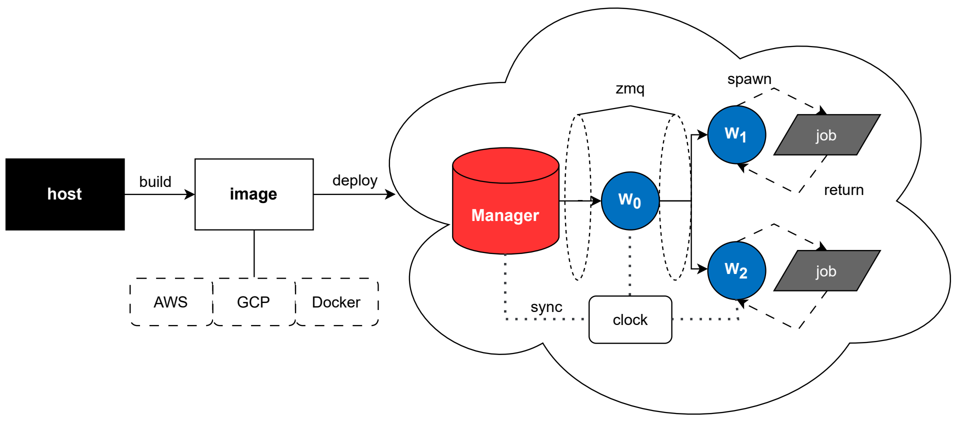


Figure 4.1: TreeBuilder's Implementation Pipeline

In this chapter, we present the software tools, methodologies, and design patterns that were employed to transform the conceptual design of TreeBuilder into a fully functional application. Figure 4.1 provides a visual overview of the core processes implemented in TreeBuilder's pipeline, which extend beyond the VM selection mechanism. These processes encompass reliable node communication protocols, the creation of reproducible infrastructure, experiment control, data logging, and other essential components. As is often the case with distributed systems, decisions regarding individual elements can directly influence the behavior of the overall system. Therefore, a holistic approach to their implementation was essential, ensuring that each piece integrated seamlessly into TreeBuilder's overarching objectives. We aimed to produce a tool that is not only effective in performance but also maintainable and extensible for future research. Coupling those ideas with TreeBuilder's design goals (section 3.1), we defined the following functional requirements:

R1. **Reproducibility**: It is crucial to ensure that procedures can be reliably reproduced across different environments. This way, we guarantee that the same infrastructure setup can be generated consistently, reducing configuration drift and manual errors.

R2. **Self-Documentation**: The system should inherently document itself through its configuration files. This approach minimizes ambiguity and makes the work understandable through its codebase, facilitating maintenance and knowledge transfer.

R3. **Version-Control-Friendly**: The ability to track changes over time is critical for development stability and debugging. With tools that integrate well with version control, changes to the infrastructure can be audited and reverted if necessary.

R4. **Flexibility**: TreeBuilder should support dynamic scaling and allow for flexible configurations, such that as the number of nodes in the system vary, it continues to perform adequately.

One key principle that will be noticed in selection of tools mentioned in this work was to prioritize standardized Infrastructure as Code (**IaC**) frameworks. IaC is a paradigm in software development that promotes the management of infrastructure through machine-readable definition files, rather than manual processes or *ad-hoc* scripts. Tools following this pattern naturally address our implementation objectives, enabling consistent (**R1**) and self-documenting workflows (**R2**), which are readily version-controlled through declarative file configurations (**R3**). In the following sections, we provide an in-depth explanation of the components that contribute to the work's overall functionality, from virtual containers and cloud services to the TreeBuilder's distributed system itself.

4.1 Infrastructure Provisioning

4.1.1 Virtual Images

We chose `Packer` [Has23a] to generate our image and container builds. Packer is a IaC tool from Hashicorp and serves as a versatile image creation tool for dockers, virtual machines and cloud instances. Similar to how Dockerfiles define the blueprint for Docker containers, packer offers a single build process for various environments through it's HCL or JSON configuration files. This provider-agnostic approach allows consistent image creation across different platforms, simplifying the modification of image dependencies and ensuring that changes are robustly perpetuated across platforms. With a shared virtual image across nodes, we can ensure an identical filesystem and software environment for all instances within the cluster.

4.1.2 Cluster Management

To deploy, destroy, and manage a cluster on the cloud, we chose to use `Terraform` [Has23b]. This application, also developed by Hashicorp, is an IaC tool that allows users to build, modify, and manage cloud or on-premise clusters through programmatically defined files. It features an intuitive proprietary executable and beyond being a *de-facto* standard in this space, Terraform enabled us to create a cloud-agnostic workflow that could be easily applied to GCP, AWS, or even locally deployed Dockers. This proved invaluable as TreeBuilder could be tested in a local environment, where mock nodes were launched as Docker containers with their own private addresses. This facilitated efficient feature additions and debugging.

Additionally, through Terraform's rich API we enable a convenient interaction with cloud provider interfaces, setting up startup scripts, transferring folders, configuring virtual machines and private networks, and much more directly through its configuration files. This then required from us little to no external adjustment with a provider's webportal and permitted a self-documenting and self-contained workflow regarding cloud management. Terraform also accepts input arguments as machine-readable JSON files, which enables us to define configurations in a highly expressive manner – adjusting IP addresses, instance names, and startup commands for each individual node. This system offers flexibility, allowing us to launch clusters generically without hard-coding node counts or roles (**R4**).

It is clear that both Packer and Terraform directly support previously established requirements. These technologies streamline the development process, where adjustments to an image or to the deployment pipeline does not warrant having to perform redundant double-efforts in the code-base for every supported platform.

4.2 Distributed System

Building on the consistent image creation and flexible deployment process established in the previous section, this section explains how the components of TreeBuilder's distributed system interact to achieve its operational goals. Using a shared virtual image and Terraform's capability to specify startup commands and configurations for each instance, the manager and worker nodes, as introduced in section 3.2.1, can be dynamically assigned their respective programs and therefore roles during deployment. Furthermore, by utilizing Clockwork's `Clock Sync` daemon [Clo23], we ensure high-precision software clock synchronization across nodes. This is essential for accurate One-Way Delay (OWD) measurements and serves as a reliable mechanism for node coordination and synchronization throughout the cluster.

4.2.1 Nodes: Manager x Workers

The next natural question is how are nodes within our distributed system implemented. The individual application logic running on every instance in our cluster is composed of one Python module named `manager`, located at the root directory of TreeBuilder's repository. When running the module, several arguments can be passed to the program as seen in appendix A.2. However, one can specify which action the node is intended to perform via the `-a or --action` flag, being that either a manager or a worker. Effectively, these indicate separate function calls that initiate the corresponding macro objects and start their main loop. Figure 4.2 illustrates the primary APIs and fields exposed by the manager and worker classes. These elements will be referenced and made clear throughout this thesis, gradually contributing to a systemic understanding of their inner workings.

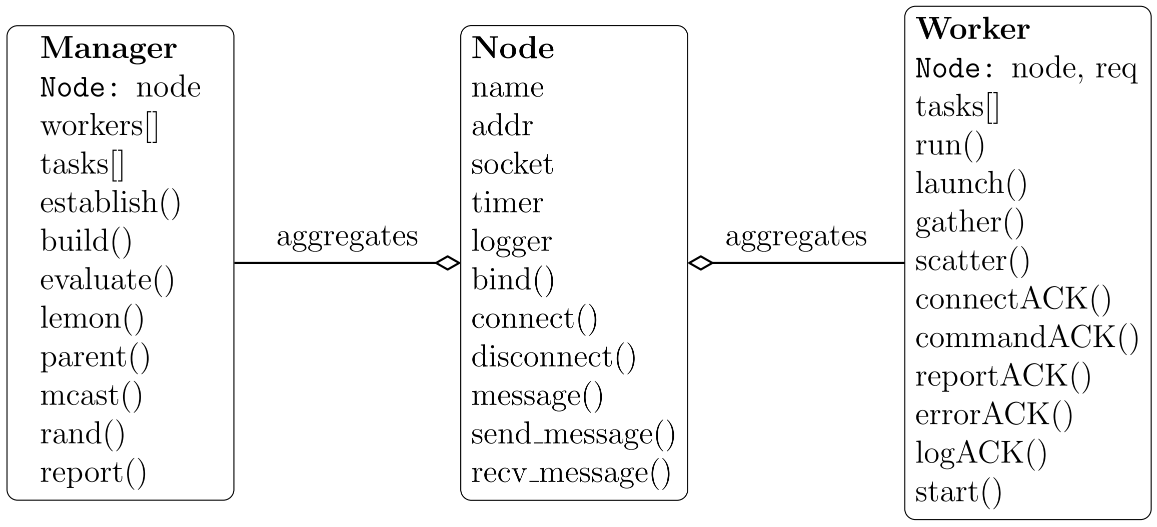


Figure 4.2: Manager x Worker x Node Classes

As per Figure 4.2, the manager and the worker classes instantiate a `Node` object. This node object leverages the `ZeroMQ` library to connect and exchange messages with other nodes in the system [Hc23]. Therefore, it becomes evident that both workers and managers possess instantiated objects that allow them to communicate. More on messaging and the custom protocol developed to meaningfully exchanged data between nodes will be explained further.

4.2.2 Messaging

ZeroMQ is a high-performance asynchronous messaging library, which is well-suited for our system due to its scalability and low-latency messaging patterns. It offers multiple communication paradigms, and abstracts the complexities of network programming by managing connections, message queues, and retries internally, enabling us to focus on the application logic. Additionally, ZeroMQ also allows for variable sizes in message buffers and paired with Google's `Protobufs` [Goo23], provides us with an invaluable manner to robustly send and receive highly expressive serialized data as struct-like objects. Protobufs is a language-neutral, platform-neutral mechanism for serializing structured data, offering efficient and compact binary formats generically described through their declarative `proto` files. Messages and their complete data structures can be seen in Figure 4.3.

Every message payload can be expressed by the macro `Message` data structure seen in the upper-left corner of Figure 4.3. A Message object also instantiates a `Metadata` and

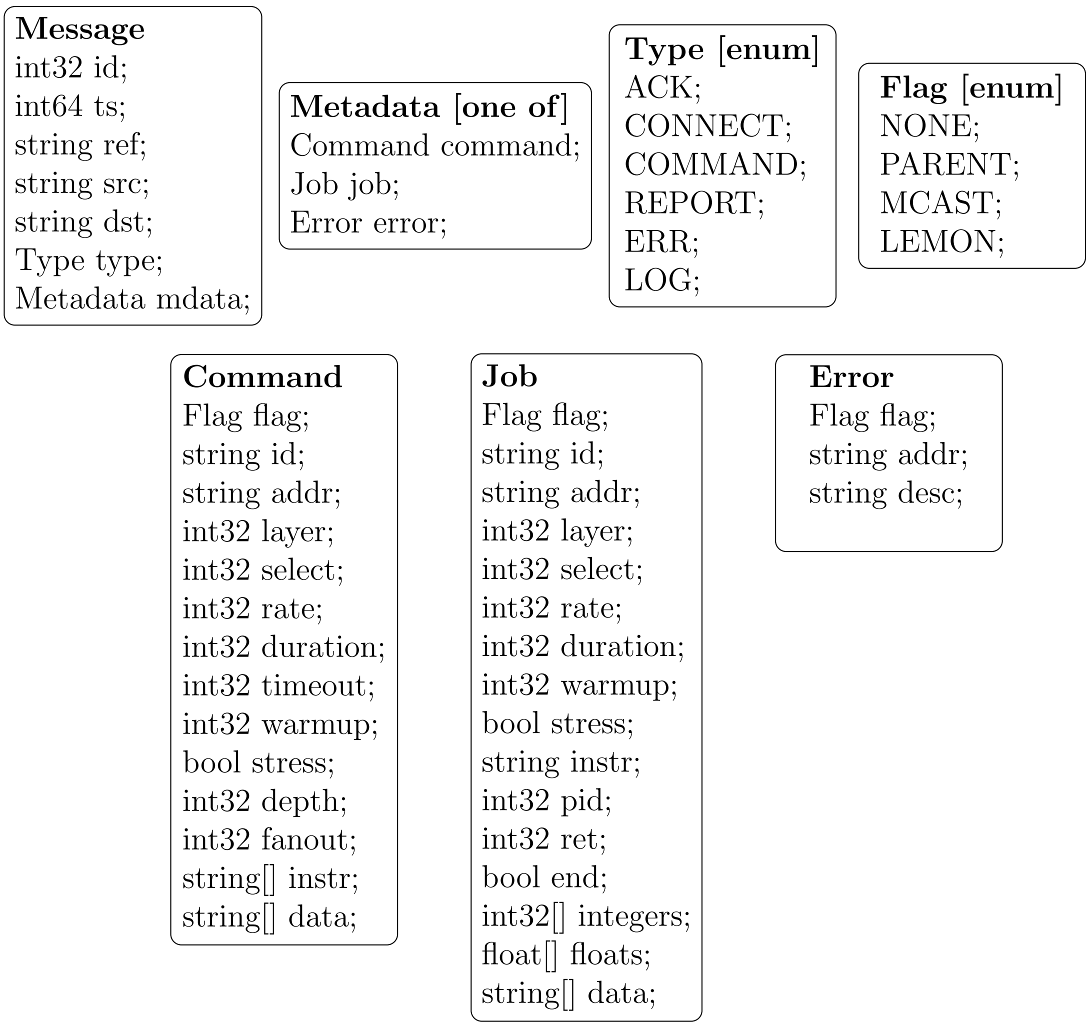


Figure 4.3: Message Data Structures

a `Type` object, respectively referenced by the `mdata` and `type` fields. Metadata is a flexible construct allowed by Protobufs `oneof` syntax, where a given field can be dynamically populated by any one of a defined list of objects. In the case of Metadata, these were the `Command`, `Job`, and `Error` structures. This facilitates a non-specific method of communication where, if needed, new objects can be added to the Metadata union-like structure and seamlessly extend the base protocol.

The Type enumeration is used to identify the category of data delivered and therefore indicate which data structure would be found in the mdata field. The possible values associated to Type are `ACK, CONNECT, COMMAND, REPORT, ERR,` and `LOG`. Furthermore, a Message object contains `src` and `dst` fields to ascertain the senders and target receivers respective addresses and ports. The `ts` field contains the timestamp in microseconds since the epoch, generated through a helper class called `Timer`, shown in appendix A.3. This is used to inform when this message was generated and help in event coordination. The `id` and `ref` fields are used mostly for internal bookkeeping and debugging purposes. The `id` field is populated by an internal tick of every node in our system. Every time a node sends a message, this tick is incremented. Therefore, this serves as an individual sequencing mechanism. Finally, the `ref` field is a string populated by the name of the sender and its target receiver separated by a delimiter character. Terraform associates each node in the system with a custom name based on the provided JSON input file, simplifying log readability and making it easy to quickly identify the purpose of exchanged messages and the nodes involved.

Socket Patterns

Having outlined the key APIs that constitute the manager and worker nodes, as well as the structure of their messages, it is now crucial to define the communication patterns or sequences that guide how data is propagated. Figures 4.4 and 4.5 illustrate these well.

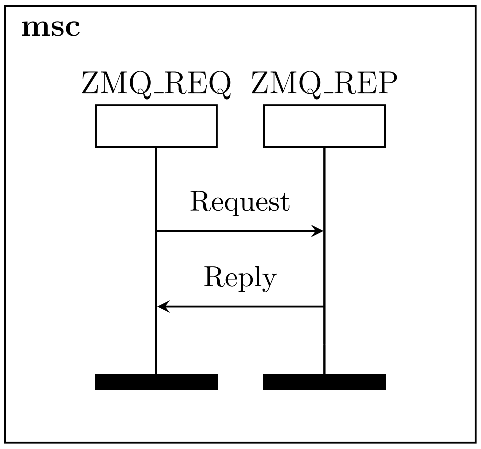


Figure 4.4: ZMQ Sequence Diagram

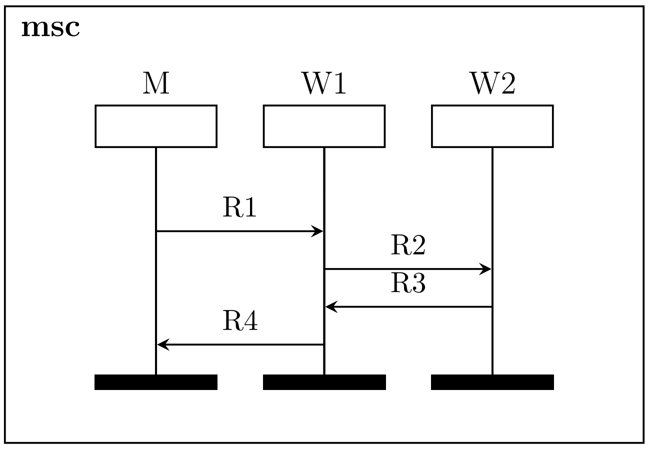


Figure 4.5: Node Sequence Diagram

ZeroMQ exposes several communication paradigms that require a variety of different socket types to realize. The most common and supported pattern is the request vs. reply sequence. In this paradigm, a `ZMQ_REQ` socket connects to a given `ZMQ_REP` socket. Subsequently, the request socket is allowed to send a message or byte sequence to the connected reply socket. As the name suggests, the reply socket is only allowed to reply to messages initiated by a request socket. Finally, the exact order of message exchange has to be respected as seen in Figure 4.4. That means, a request socket can only send a new payload after the reply socket has answered. Conversely, a reply socket can only receive new messages once the previous one has been replied to. Even though such a protocol seems rigid, it does guarantee connection maintenance as sockets await for replies and makes sure to correctly aggregate data contiguously on the receiver's end. Other socket types were available within ZeroMQ's library, but mostly did not provide a superior trade-off in simplicity and feature usability.

Building on the request and reply paradigm, the manager's node object is configured as a `ZMQ_REQ` socket, while all workers' sockets are of the `ZMQ_REP` type. This layout already enforces the design, where a manager node is an active initiator, whilst workers react to events in a passive manner. However, as seen in Figure 4.2, the worker class has, along with its main `Node` object (node), another one of the same type named `req`. Even though all workers instantiate their main sockets as reply sockets, they also embody an additional request socket to allow for active connections and sending of data. This is shown in Figure 4.5, as the manager node initiates a message with a worker node named `W1`, which in turn actively connects and corresponds with a second worker node `W2`, before replying to the manager's initial request. This comes heavily into play during the scattering of work and querying for results.

4.2.3 Communication Protocol

Understanding the mechanism in which messages are structured, the fields they contain, and their patterns of exchange, we now will go through the established customized protocol in which nodes interact. The message types are delineated through the `Type` field of a `Message's` payload, as shown in Figure 4.3.

Type: CONNECT

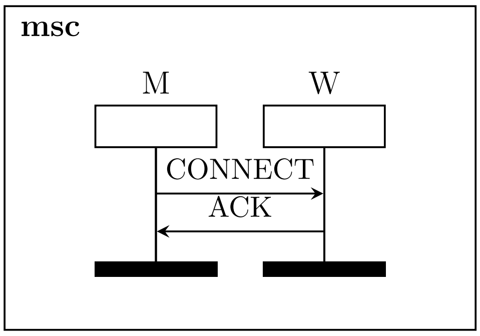


Figure 4.6: CONNECT Sequence

This is the most basic form of data exchange, where any given node in the system establishes connection with another via a message greeting of the `CONNECT` type (Figure 4.6). As only workers accept incoming connections, they are made to reply through the `connectACK()` function as shown in Figure 4.2. This is done via a creation of a new message object through `message()`, populating accordingly the `src, dst, ref` and `ts` fields, and finally setting the reply's `type` field to `ACK`. Every response message's type field is set to `ACK`, this helps us identify if messages are requests or replies, and if not properly done will cause the system to go into fault. `CONNECT` messages are sent out to all workers during the manager's boot-up routine via the function `establish()` and are required to verify a healthy cluster before starting our experiment. Additionally, workers also establish connections with one another through their `req` objects and do so as a prologue to some form of additional request or data transfer.

Type: COMMAND

Expanding on the concept of nodes being able to establish a connection with one another, the first mechanism of a more complex form of data transfer are `COMMAND` messages.

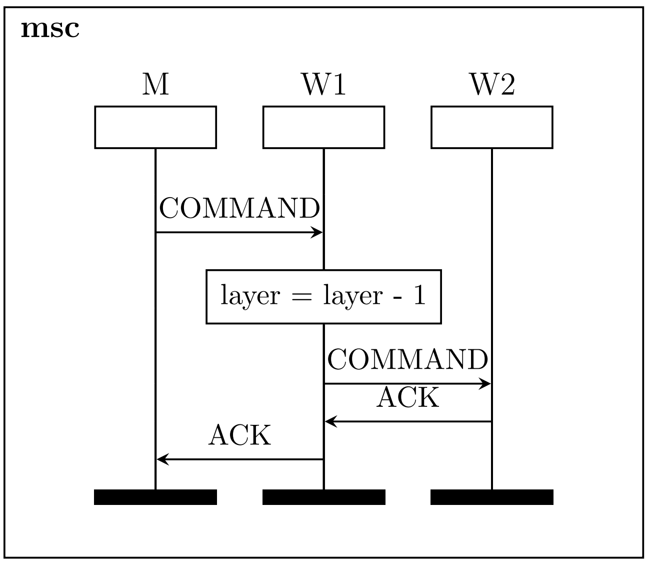


Figure 4.7: COMMAND Sequence

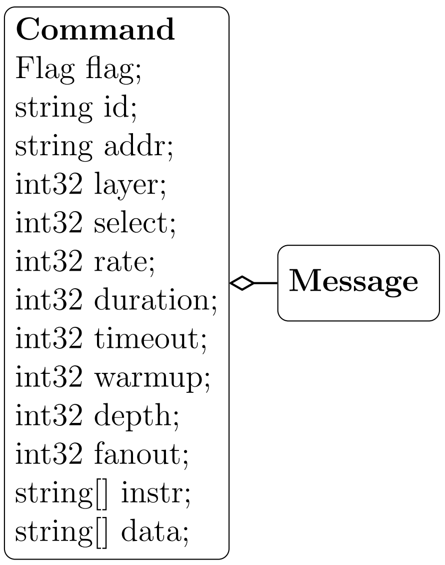


Figure 4.8: Command Structure

Figure 4.7 displays a scenario of a command message flow across the system. To facilitate the comprehension of this typical exchange, Figure 4.8 reiterates the `Command` data structure that is found within a Message's `mdata` field. Given our design and socket types, the manager is able to to send commands to workers, which depending on said message's structure, might have to relay additional commands further. Therefore, workers are the only nodes allowed to receive commands.

First and foremost, we have the `Flag` enumeration object, referenced within the `Command` payload as `flag`. This enumeration describes to the receiver what kind of `COMMAND` this is. As seen in Figure 4.3, these can be either `PARENT`, `MCAST`, `LEMON` or `NONE`. This will directly instruct a worker what executable should be run, as well as how to correctly process this message. More on the different available command executables and their processing will be shown in section 4.2.4.

A crucial mechanism in this exchange is the `layer` field. It is set by the manager during the command's inception and describes to a worker its rank within this distributed task. If the `layer` value is above zero, the worker understands that it must, before executing the described command, inform other workers on their respective roles in this effort. The `instr[]` and the `data[]` string arrays contains, respectively, the necessary CLI instructions and the corresponding addresses for the current and next layer of workers. Any given CLI instruction is run as a separate process within a VM's operating system through Python's `subprocess` library. How a worker parses `instr[]` and `data[]` to properly extract the needed instructions and addresses for the next layer will be explained in depth in section 4.2.4. Before sending the corresponding messages to the next layer of workers, the current worker decrements the layer field. Only after the next layer of workers successfully reply, will the currently focused worker run its job and reply its status to its sender. Workers whose layers are zero, simply directly run their command and reply with its status. This mechanism proves invaluable both for correct coordination of events, as well as a way to naturally express distributed tasks in a tree-like fashion.

As per Figure 4.2, these actions are taken care of through the `commandACK()` function call. There, a worker will `scatter()` command messages, if they are assigned non-zero layers, process their responses and by calling `launch()`, both `run()` their individual instruction and afterwards `gather()` the downstream job results if needed. The `select, rate, duration, timeout, warmup` and `depth` fields are parameters associated to the described command and will come into play later.

However, it is not yet clear the format in which workers reply to their issued commands. Beyond populating the major fields in `Message` and setting the message type to `ACK`, workers fill their reply's `mdata` field with a newly created `Job` object. With help of Figure 4.3, many of the fields within `Job` are identical and in fact taken from the received `Command` structure upon its arrival. Most importantly for now, the `Job` object carries the same `id` as its command counterpart, holds relevant metadata associated to the to-be-ran instruction and a copy of it is sent as an acknowledgement reply to the command issuer. That means, within the `ACK` message reply performed by workers who have been issued a command, contains a dummy copy of the internal `Job` structure created to execute the corresponding instruction. This is used for result probing and keeping tabs on tasks across the system, as it will be explained in the following section on `REPORT` messages.

Finally, the `addr` field is needed to verify that the recipient is the intended target worker and causes the system to fault otherwise. The `id` field is generated by the manager and is a unique random string associated to this distributed task. This is used for both debugging and logging purposes.

Type: REPORT

At this point our system has a manner in which it can connect to arbitrary nodes and distribute a select type of commands across a cluster in a layer-like fashion. Additionally, it is also clear that workers reply to issued commands with a `Job` structure within the message's `mdata` field. A `Job` structure is a vessel that summarizes any running job in our system and it is created as `COMMAND` messages are distributed to workers in a cluster. They are held by the worker that executes a described command and a copy of it is sent to the node (manager or worker) that submitted it, as they are then responsible for querying or probing for its results. As seen in Figure 4.10, a `Job` payload contains several meta-parameters associated to the command dispatched to a worker such as `select, rate` and `duration`. Their use are explained in further detail in section 4.2.4.

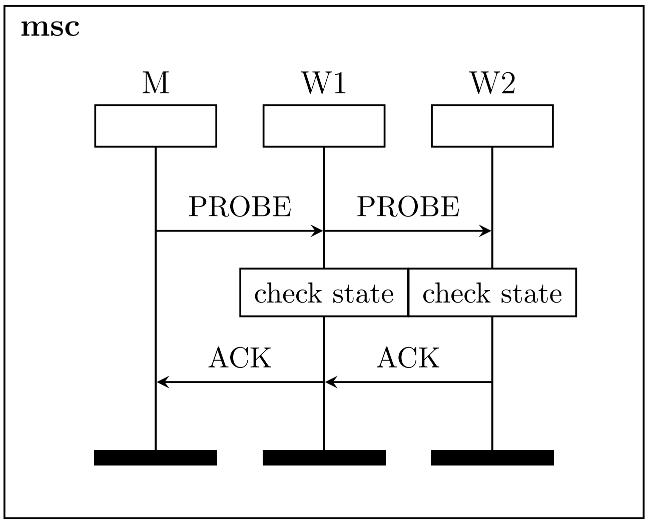


Figure 4.9: REPORT Sequence

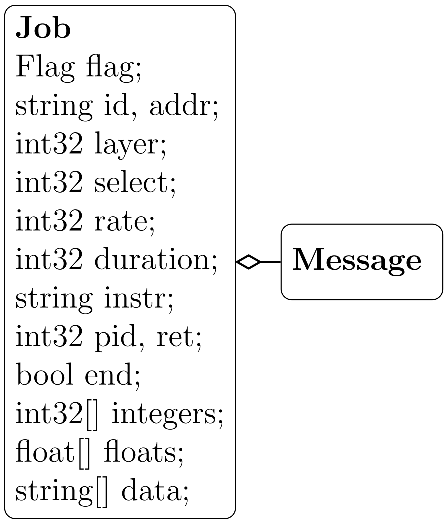


Figure 4.10: Job Structure

Furthermore, the `Job` structure holds the instruction to be run by a worker. As per Figure 4.2, the worker spawns a separate thread via `launch()`, which runs the instruction as a process in the VM's system, awaiting for it to return from execution. Once finished, the results are parsed from the processes' standard out and its return code and results are respectively stored in the `Job's ret`, `integers[], floats[]` and `data[]` fields. How the ouputted stream of an executable populates these fields is informed by the initial command's value in `flag`. These values, being either `PARENT`, `MCAST`, or `LEMON`, can directly be mapped to different classes that, among other use-cases, serve to parse a `Job's` instruction's output and correctly update its fields. More on this will be seen in section 4.2.4.

Having established the purpose of a `Job` structure and the values it holds, we now explain the `REPORT` message flow as illustrated by Figure 4.9. Once distributed commands are issued to a subset of workers, those who were of non-zero layers, as well as the manager, immediately start sending `REPORT` messages to their associated downstream node(s). These are done simultaneously and in specified time intervals, which is the process of probing for results that has been mentioned in section 3.2 on TreeBuilder's design. Given that the correct reply to `COMMAND` messages is a duplicate of the internal `Job` structure being processed, `REPORT` messages, when sent to the original job owners, include this same `Job` structure within their `mdata` field. This is done to inform the receiver, which job is being queried for since they hold unique ids generated by the manager. The receiver updates the `REPORT` message's `Job` structure with the latest state of the corresponding internal `Job` it currently is processing. In case of a `id` mismatch the system enters into fault. As a job finishes, its results parsed and the associated `Job` structure finally populated, a querying node will receive a job's final summary and react to it accordingly. The status or outcome of the deployed job will be reflected by its `end` and `ret` fields, while their results will strategically be placed in the payload's `integers[]`, `floats[]` and `data[]` arrays. As this is done recursively throughout the layers of a distributed task, the manager node will eventually aggregate the entirety of results associated to a given job.

Type: ERR

Error-handling and fault recovery are fundamental characteristics to distributed systems. There has been mentions of nodes faulting given specific scenarios. `ERR` messages are used in TreeBuilder to express a fault as a node processes an incoming message. This is seen in Figure 4.11.

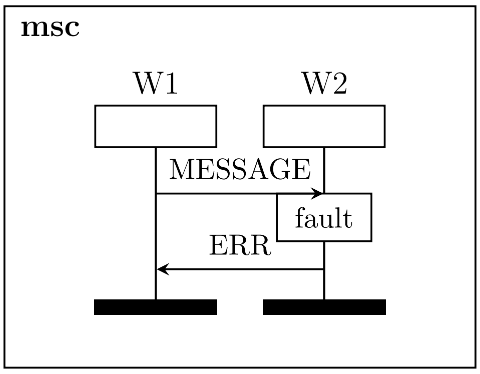


Figure 4.11: ERR Sequence

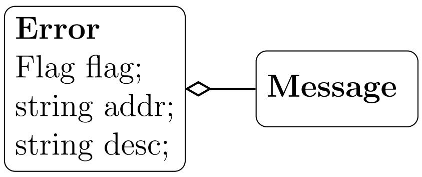


Figure 4.12: Error Structure

As the recipient of a message encounters an unexpected outcome to the computation following it, the receiver will create an `Error` object with a description of the issue in its `desc` field. This is appended to the message's reply and its `type` field is set to `ERR`. This can occur in several scenarios such as when a node mistakenly queries a job that either does not exist or is not owned by the target receiver. It is important to note that, errors during a job's instruction execution are already reflected through the `Job` data structure and its `ret` field. Therefore these do not need to be expressed via a `ERR` message. Finally, `ERR` messages are mostly in place for sanity checks when nodes dispatch requests to one another and are not implemented as a full-fledged fault recognition or fault-recovery mechanism.

Type: LOG

The `LOG` message type is used to request nodes in the system to flush their logs. This scenario is shown in Figure 4.13. The `Node` class used by all workers contains a `Logger` object. This data structure is a wrapper around Python's standard `Logging` library and is used to asynchronously write JSON-like dictionaries both to standard out and into log files. These are pushed to cloud storage by the end of an experiment and contain every sent and received message from any given node, as well as relevant information regarding events during its runtime. `LOG` messages are sent either when the manager realizes a node has faulted in its behavior or at the very end of a successful experiment.

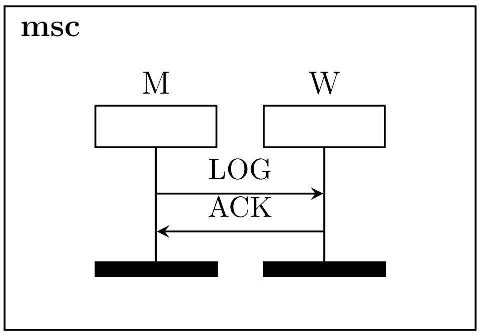


Figure 4.13: LOG Sequence

4.2.4 Collective Tasks: Jobs

Having established the foundation for image creation, cloud deployment, manager and worker nodes, message structures, and the protocol for data transfer, we now turn to the types of collective tasks or jobs performed by TreeBuilder. In Section 3.2 on TreeBuilder's design, we described various job types, which are directly mapped to the `Flag` structure in our messages (Figure 4.3). These flag values – `PARENT`, `MCAST`, and `LEMON` – indicate the specific job type referred to by a sender and influence how `COMMAND` and `REPORT` messages are handled. In this section, we will delve into the how these collective jobs are implemented and how they relate to the previously mentioned components.

Tasks

The concept of tasks are introduced in Figure 4.14 via the `Task` superclass, as well its relationship to the manager and worker nodes. We knew that there would be several similarities as to how different job types are handled and processed. So, by leveraging a factory method software pattern, we developed individual class objects for each of the distributed jobs TreeBuilder needs. All of whom inherit the methods and fields from `Task` and overwrite functions that are individual to them. Furthermore, this encapsulation method allows for appropriate logic segmentation and easy extension of either current tasks or newly added ones. `Task` objects are used by the manager and worker nodes, where each may own a varying number of such instances within their `tasks[]` field. `Tasks` are utilized to both (i) hold a job's state and (ii) interact with its internal data across different stages of a distributed job's deployment pipeline. They are a crucial mechanism in TreeBuilder's operation and are involved in building messages, scattering commands, parsing output and evaluating results. Therefore we will outline the principal methods seen in its superclass and correlate them to the larger phases – `Build, Scatter, Parsing, and Evaluation` – of a collective job employment.

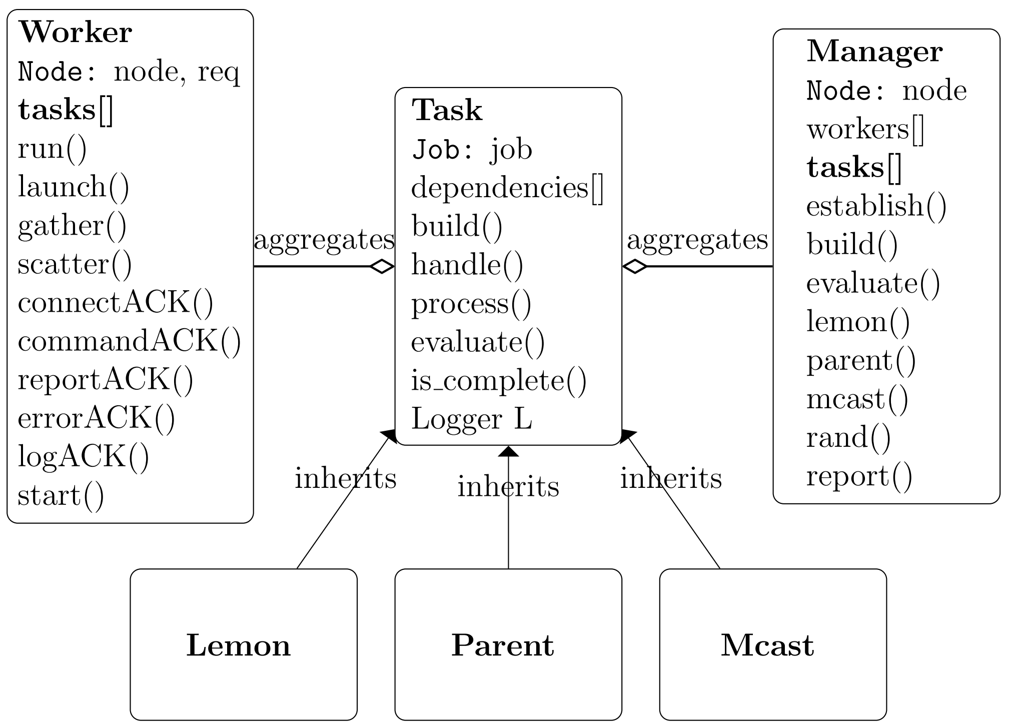


Figure 4.14: Manager, Worker, and Task Class relationships

Build Phase. A `Task` object is created by the manager at the first stage of a distributed job's employment. There, depending on the job type, the manager will either invoke the `parent()`, `mcast()` or `lemon()` calls. This internally creates a corresponding `Task` object – `Parent`, `Mcast` or `Lemon` – that is used to both refer to said job and build the initial `COMMAND` message needed to deploy it. The manager uses the `Task` object's `build()` function call, which formats the CLI instructions executed by each worker node and their addresses within the fields of the outputted `COMMAND` message. The underlying executable binaries are the core of the different task objects and their their arguments are shown in appendix A.2. These can be mapped to the established experiment parameters that will be spoken on in section 4.4. Once the initial `COMMAND` message is ready, the manager sends it down to the given job's leader (see section 3.2).

Scatter Phase. A worker or the job leader processes the `COMMAND` message through the `commandACK()` call. It will, depending on its layer rank, `scatter()` further commands downstream, capture their replies, and `launch()` a process to execute their instruction. Finally, they then reply to their original sender with a `Job` structure copy of this internal task, as the given process is successfully spawned as a separate thread. In order to parse a `COMMAND` message's `data[]` and `instr[]` fields to correctly identify the addresses and instructions to be sent further, the receiving worker will also create an internal `Task` object based off of the command's `Flag` value and invoke its `handle()` method. In a simple single-layer scenario, such as for a `PARENT` job, this process is straightforward. However, when describing a tree structure of generic depth and fan-out, additional logic is needed to properly extract the data relevant to the current node and manipulate the lists of instructions and addresses for each child in the next layer. This is where the `depth` and `fan-out` parameters, found in both the `Command` and `Job` payloads, become critical. In summary, the manager compacts the tree-like description of instructions and their corresponding addresses into lists within the initial `COMMAND` message payload. On a layer-by-layer basis, workers extract the subtree structures from the received message and create new `COMMAND` messages for the next layer. The procedure of extracting, from a tree in a list form, the subtrees corresponding to every child of the current node is referenced in this work as slicing. The data structures used to both express a tree and manipulate its data can be seen in appendix A.3.1. Once a received `COMMAND` message is correctly distributed to the next nodes, a worker will start its execution in a concurrent manner.

Parsing Phase. Apart from extracting and formatting command instructions, `Task` objects are also needed to process the output of their underlying ran executable. This is done by the `process()` function call. Again, the notion of individualized classes are proven to be useful as it gives freedom to easily add new tasks to TreeBuilder and keep their processing customized and encapsulated. This occurs after a given job finishes its execution, where the supervising worker node automatically invokes the given `Task's` processing function within the launched thread. Given the format of the text outputted to standard out by the `parent()`, `mcast()` or `lemon()` executables, each different `Task` inherited object will parse said data and fill it accordingly in the `integers[]`, `floats[]` and `data[]` fields of a `Job` structure. This `Job` data structure is held by the overarching `Task` object and when queried for by the manager or another worker node is appended to the reply of a `REPORT` message. As explained by TreeBuilder's probing mechanism, the manager eventually receives a final `Job` structure summarizing the collective-task's deployment.

Evaluation Phase. This is the last stage of a task's pipeline and is done by manager after results are gathered. The very first moment of the build phase had the manager create a `Task` object used to keep tabs on the deployed job, as well as construct the first message sent to the job's leader. Now, that the results are available, the manager invokes the `Task's evaluate()` method, which is individual to each of the different tasks in our system, and takes in the replied `Job` structure and the currently used strategy **S** as its arguments. In the case of `MCAST` jobs, the results are logged into the filesystem and stored internally for future use. However, how the tasks related to `PARENT` and `LEMON` jobs process the data is a separate concept that will be explained in the following sections. The core of TreeBuilder's straggler optimization technique and our re-implementation of LemonDrop are directly based off of this stage of a tasks pipeline.

4.3 VM Selection Methods

We have outlined TreeBuilder's selection algorithm in the section 3.2.3. Coupling that with LemonDrop's adapted FAQ routine and the task deployment control flow explained previously, we now show how these optimization methods are applied within our system. The final lifetime of a task involves parsing a `Job` structure found in the `mdata` field of a `REPORT` message. This is done via the `Task's evaluate()` method. This function processes the data and creates, per VM, a statistical summary of the OWD latency perceived by that machine during a given job's duration. This summary is represented internally through a dictionary data structure called `ItemDict`, as seen in Figure 4.15. One can notice that a `ItemDict` structure contains key metrics of the described job for each of the involved VMs. These are the 90th, 75th, 50th and 25th OWD percentiles, as well as their mean and standard deviation. Additionally, we count the number of received packets to take into account drop percentage, if needed.

ItemDict
addr: str
p90: float
p75: float
p50: float
p25: float
stddev: float
mean: float
recv: int

Figure 4.15: Job Summarization Structure: ItemDict

In the case of `LEMON` jobs, the manager needs the `ItemDicts` regarding each of the VMs in the cluster to form the required OWD matrix for LemonDrop's procedure. As described in section 2.1.3, the OWD matrix is composed of the pair-wise median latency values among all nodes in the system, and the load matrix, is the normalized number of requests between nodes per unit of time. The latter is easily derived off of a given packet rate and a tree's defined depth and fan-out. Through our recreated LemonDrop's FAQ algorithm, shown in appendix A.4.1, we are able to apply its procedure onto the managers gathered data and obtain a mapping scheduling of VMs for a given multicast tree.

In the case of `PARENT` jobs, all of these fields can be used in the evaluation of the final `Job` structure. Now, we move onto TreeBuilder's selection method and its application of strategies. These directly build upon the summarized data structure shown in Figure 4.15 and explain how their individual entries influence the system's outcome.

4.3.1 TreeBuilder Strategies

As described in section 3.2.3, TreeBuilder's design embodies the concept of a strategy **S**, which informs how our tool is supposed to rank and select VMs off of the results of `PARENT` jobs. These results are structured as a series of `ItemDicts` that hold key statistical metrics regarding a VM's perceived OWD latency during a tasks duration. A strategy is nothing more than a lambda function indexed by a string value that tells the manager how to score a VM's `badness` from a given `ItemDict`. We currently expose three different VM selection strategies:

Strategies	**Scoring Heuristic**
WEIGHTED	(p90 x 0.3) + (stddev x 0.7)
P90	(p90 x 1)
P50	(p50 x 1)

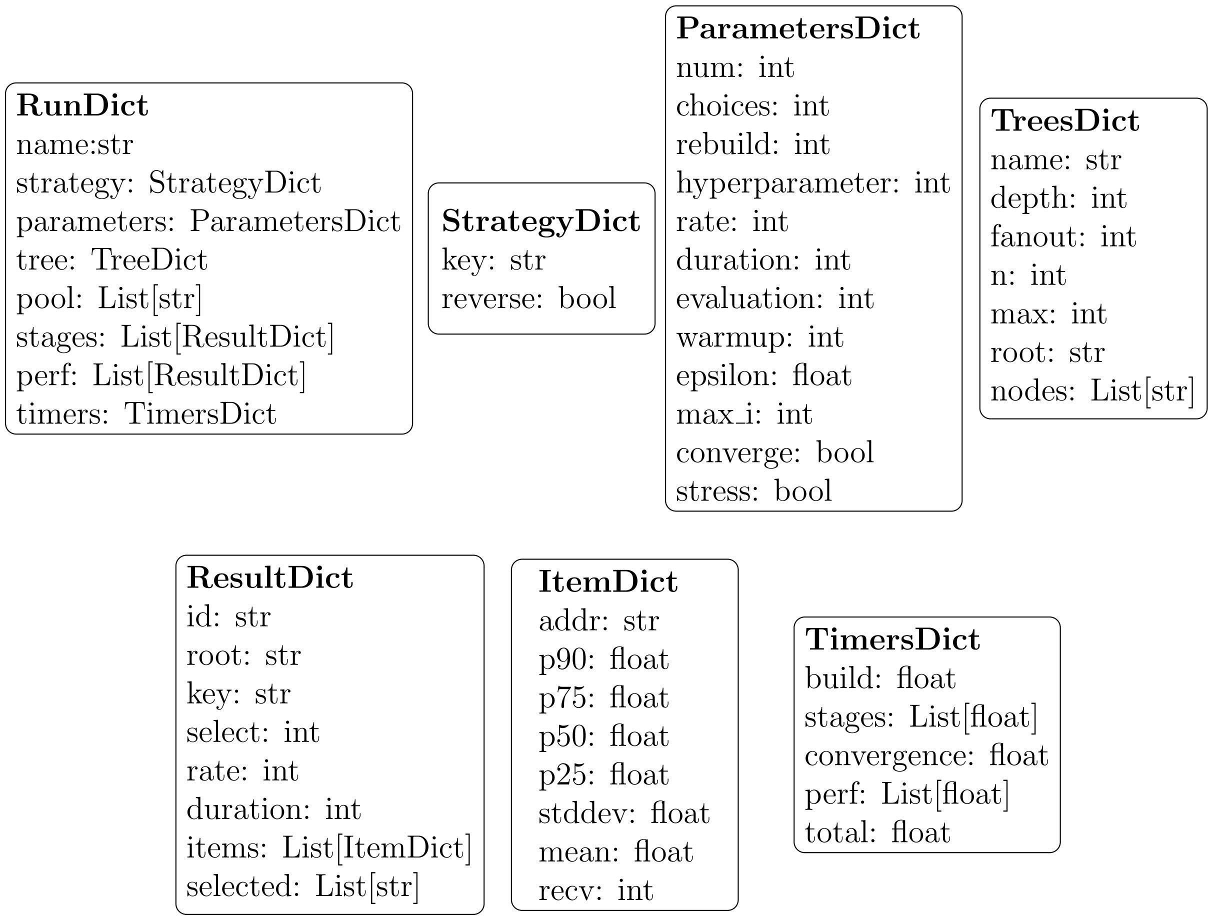
Table 4.1: TreeBuilder's strategies and scoring mecanisms

Now, if a given tree is employed using the `P50` strategy, it will rank VMs based off of their overall median latency. If we instruct the manager to optimize a given tree for the better, it will select machines that have the lowest `P50` score. Conversely, when instructed to optimize for worst nodes, the logic is reversed. The idea here is to permit generic strategies that leverage the keys exposed by an `ItemDict`. This allows for easy extensions to current methods and varying attempts of a given heuristic. The same notion is maintained for the `P90` and `WEIGHTED` strategies. In the case of the `P90` method, its ranking is directly dependent on a VM's OWD latency 90th percentile value. Finally, for the `WEIGHTED` strategy, we show our first attempt of a weighted sum approach. This method is supposed to optimize for VMs with low 90th percentiles and consider their variancy through the weight placed on the standard deviation key.

The objective is to visualize how simple metrics can affect the final system, as well as enable an infrastructure that allows for precise refinement of our straggler optimization technique with little change to the underlying codebase. There is room for an immense combination of individual keys and weights to express a scoring mechanism, but within the timeframe of this work we were only able to experiment with the mentioned strategies in table 4.1.

4.4 Experiment Control

Several parameters are used throughout the procedures implemented in TreeBuilder. One can specify high-level values such as the depth and fan-out of a tree, the strategy it should employ and how many different root selections it should go through. Apart from that, lower-level variables can be set for the duration of specific jobs, packet rates, and much more. To allow for both a manner to document a given experiment's global settings and propagate several parameters from an experiment deployment down to TreeBuilder's runtime, we used nested dictionaries as seen in Figure 4.16.



RunDict
name:str
strategy: StrategyDict
parameters: ParametersDict
tree: TreeDict
pool: List[str]
stages: List[ResultDict]
perf: List[ResultDict]
timers: TimersDict

StrategyDict
key: str
reverse: bool

ParametersDict
num: int
choices: int
rebuild: int
hyperparameter: int
rate: int
duration: int
evaluation: int
warmup: int
epsilon: float
max_i: int
converge: bool
stress: bool

TreesDict
name: str
depth: int
fanout: int
n: int
max: int
root: str
nodes: List[str]

ResultDict
id: str
root: str
key: str
select: int
rate: int
duration: int
items: List[ItemDict]
selected: List[str]

ItemDict
addr: str
p90: float
p75: float
p50: float
p25: float
stddev: float
mean: float
recv: int

TimersDict
build: float
stages: List[float]
convergence: float
perf: List[float]
total: float

Figure 4.16: Experiment Typed Dictionaries

Using Python's `TypedDict`, we serialize and deserialize data structures into `JSON` files, ensuring type and structure consistency. This allows for symmetricity in experiment configurations. Additionally, since Terraform accepts `JSON` files as input, this meta-configuration, referred to as a `schema` in this work, serves a dual purpose: it provides Terraform with the necessary values for cluster deployment – such as node names, roles, and IP addresses – and supports experiment control. Finally, Terraform packages the project repository, along with the generated schema, into a compressed archive, which is then uploaded to the cloud. Each node in our cluster downloads and extracts this archive, highlighting the importance of expressing TreeBuilder's experiment parameters in a machine-readable format that can be reliably parsed across the system.

We now outline the main elements in how a `Run` is described within TreeBuilder. However, one must keep in mind that an experiment constitutes an array of `Runs`, as each element references only how one given tree is constructed and evaluated. The top level dictionary `RunDict` contained all entries needed to specify the conditions and parameters of a tree's construction, it's produced results, and the generated tree configuration. The `StrategyDict` contains the string representation, named `key`, to instruct TreeBuilder which strategy is to be employed. Most importantly, the `ResultDict` is used to express a summary of any ran job, containing within itself a list of `ItemDict` dictionaries, which distill the key statistical metrics, per VM, concerning any of the mentioned distributed jobs. The `ResultDict` is seen both in the `stages` and `perf` fields of `RunDict`, as they respectively capture the output data per parent job used in the tree construction and the output data per tree performance evaluation. Finally, the `TimersDict` consists of timed intervals of specific phases in TreeBuilder's operation.

Finally, through the `deploy` module as shown in appendix A.2, we can create an experiments complete `schema` and launch our cluster with the manifested experiment runs and their parameters. We are able to define how many different roots are to be selected per tree, how many evaluation runs are we to perform on a given system, the lower-level variables to the `PARENT` and `MCAST` jobs, as well as which strategies do we aim to utilize. How `schemas` are parsed by the manager can be seen in appendix 4.4.

Chapter 5

Evaluation

5.1 Experimental Setup

In our tests, we deployed a cluster on Google Cloud's Platform via Terraform [Has23b]. Cloud VM instances were of *"c2d-highcpu-8"* type. Each machine offers 8 virtual AMD Milan CPU's, 16GB of Memory and 16Gbps of Network Bandwidth. The parameters for the main experiment presented in this work are summarized in Table 5.1.

Parameter	**Value**
D – Tree Depth	3
F – Tree Fan-Out	2
K – Parameter	4
P – Pool Size	25 VMs
Duration Parent Job	10 sec
Duration Mcast Job	30 sec
Packet Rate	10k packets/sec
Number of Runs	2
Number of Roots	2

Table 5.1: Experiment Parameters

Trees	**Strategies**
BEST	P90, P50, WEIGHTED
WORST	p90
RAND	–
LEMON	$\epsilon = 1 \times 10^{-4}, n = 1 \times 10^{3}$

Table 5.2: Tree Types

Strategies	**Scoring Heuristic**
WEIGHTED	(p90 x 0.3) + (stddev x 0.7)
P90	(p90 x 1)
P50	(p50 x 1)

Table 5.3: Strategy Score Calculation

In addition to LemonDrop's benchmark, TreeBuilder constructs three types of trees: `BEST`, `WORST`, and `RAND`. As their names imply, `BEST` and `WORST` are optimized in opposite ways, while `RAND` is formed by randomly selecting nodes from the pool. In the case of `BEST` trees we apply three different strategies as to display how these metrics affect the final system. We will focus mostly on the `WEIGHTED` heuristic, as it balances out characteristics we strive for. Two different root nodes are selected and shared among all tree construction methods. Finally, each tree is run twice to ensure statistical robustness, with the worst-performing run selected for examination. The next sections present the results of this experiment.

5.2 Results

The primary techniques used to analyze the collected data for evaluating a tree include: (i) plotting the Cumulative Distribution Function (**CDF**) of the OWD from the root to the leaves of the tree, (ii) examining the Time Series Plot (**TSP**) of a specific metric over a given time period, and (iii) summarizing key statistics of the tree's worst-performing leaf. We now proceed to answer the following questions:

Q1. How large is the impact of including stragglers in the system?
Q2. What effect do different strategies have on the overall system performance?
Q3. How does root selection affect the final system?
Q4. How does TreeBuilder's method compare against LemonDrop?

5.2.1 Straggler Impact

BEST-WEIGHTED vs. WORST

The first natural challenge is to quantify the impact of potential stragglers in a cluster (**Q1**). Given Jasper's multicast hold-and-release mechanism is based off of high-percentile OWD latencies of its recipients, comparing the worst receivers between two systems is a valuable metric. Table 5.4 directly contrasts the worst performing leaf's key OWD metrics from the BEST-WEIGHTED and WORST-P90 trees. The WORST tree's leaf node (W_16), observes a OWD latency at the 90th percentile of 498 μs, which is a 39.11% latency increase from BEST-WEIGHTED least efficient node (W_9) at 358 μs.

	p90	**p50**	**mean**	**stddev**	**receiver**
BEST-Weighted	358 us	247 us	261.4 us	78.95 us	W_9
WORST-P90	498 us	318 us	353.66 us	352.29 us	W_16

Table 5.4: Worst Leaf Receiver Key Statistical Metrics per Tree

Furthermore, Figures 5.1 and 5.2 show the OWD latency CDF of the respective BEST-WEIGHTED and WORST-P90 trees, regarding each of their leaves, throughout the 30 second evaluation period. The worst receiver is highlighted per tree in contrast to the rest of system. These show that the BEST-WEIGHTED tree displays more tightly grouped receivers CDFs in comparison to those of the WORST tree. The sparse receiver CDF placement in the WORST tree indicates greater differences in the observed latencies among its receivers, which suggests greater variability in its network paths. This is caused by the existence of stragglers in one's system. Apart from a significant difference in the WORST tree's least performant node, Figure 5.2 shows that almost half of its receivers are right-shifted considerably in comparison to its baseline. This indicates a systematic deficiency in network performance, as opposed to a single undesirable outlier. Finally, since the WORST tree inversely ranks VMs by their OWD 90th percentile values, the gaps between receiver CDFs are most notable at higher percentiles.

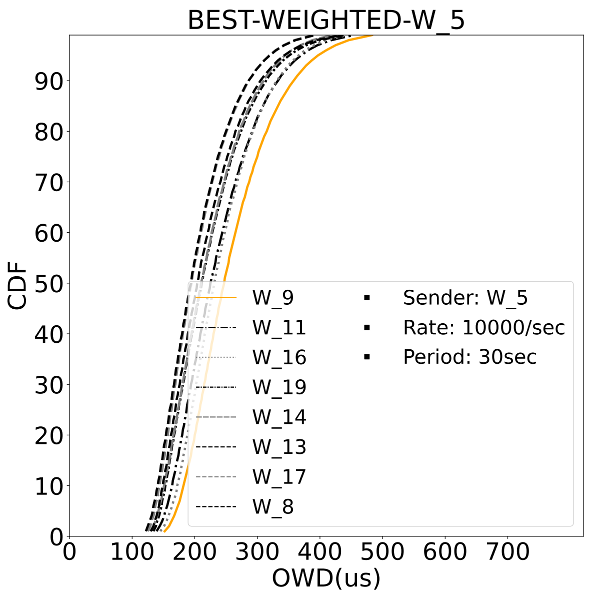


Figure 5.1: OWD Latency CDF

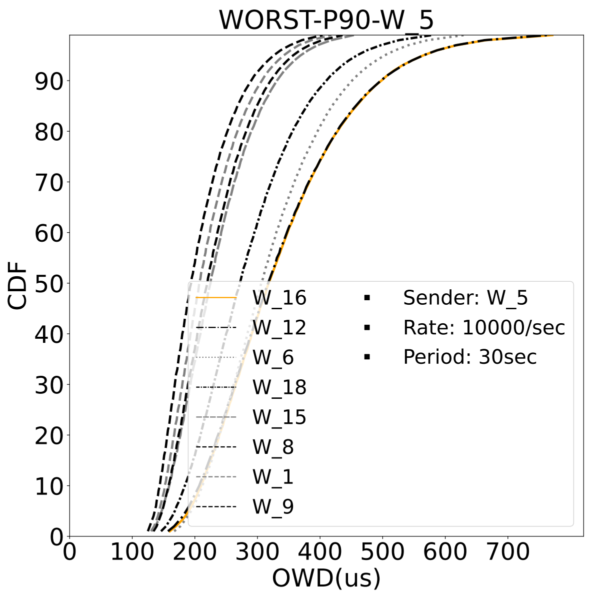


Figure 5.2: OWD Latency CDF

Figure 5.3: **BEST-Weighted Vs. WORST Tree Comparison.** `BEST-WEIGHTED` shows more tightly grouped receiver CDFs and overall lower OWD latency among leaf nodes.

Moreover, we can see from Figures 5.4 and 5.5, the OWD latency standard deviation per leaf throughout this evaluation period.

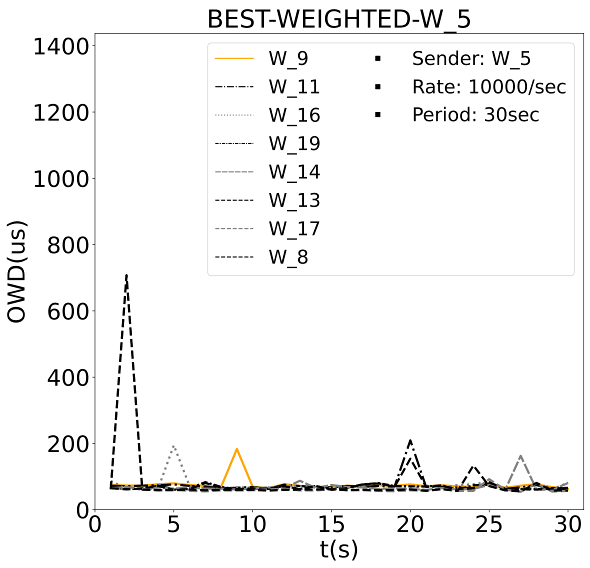


Figure 5.4: Standard-Deviation TSP

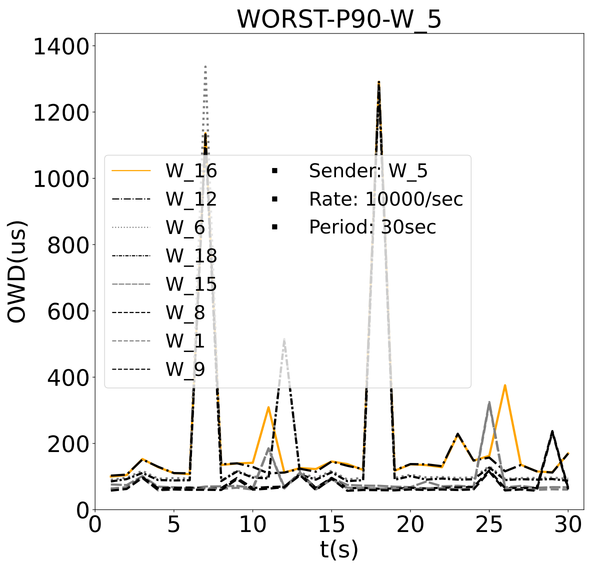


Figure 5.5: Standard-Deviation TSP

Figure 5.6: **BEST-Weighted and WORST Tree Variances.** Optimizing for selecting worst performing nodes creates a system significantly more volatile, as seen through the frequent and higher latency spikes of the `WORST` tree.

The standard deviation is calculated within every second interval of the tree's examination, which is why it is referenced as a time series plot. The present data substantiates our prior statement, as we can see a significant difference in variance between the `BEST-WEIGHTED` and `WORST` trees. The frequency and the magnitude of latency spikes are more present in the `WORST` tree's leaf nodes. Furthermore, the `WORST` tree highest standard deviation value of 1337.13 μs is a 89.13% increase in comparison to that of the `BEST-WEIGHTED` tree, at 707.76 μs.

RAND-Tree Baseline

We also present the results of the `RAND` tree in Figures 5.7, 5.8, and 5.9. The purpose of `RAND` is to establish a baseline where a user does not apply any intelligent selection to the nodes in a given cluster.

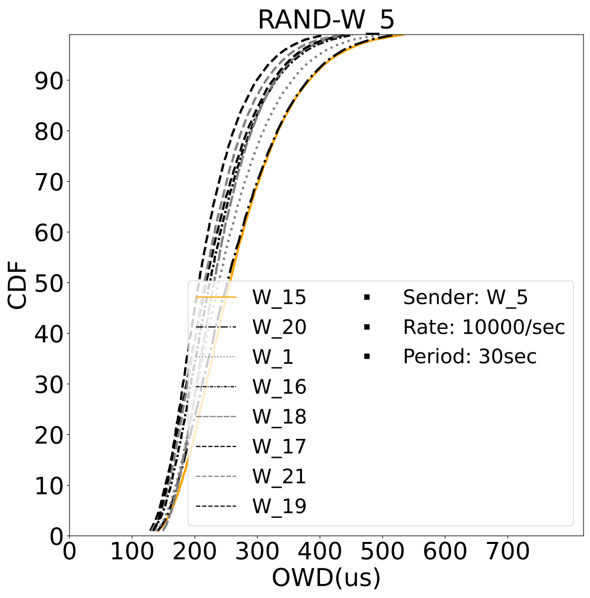


Figure 5.7: OWD Latency CDF

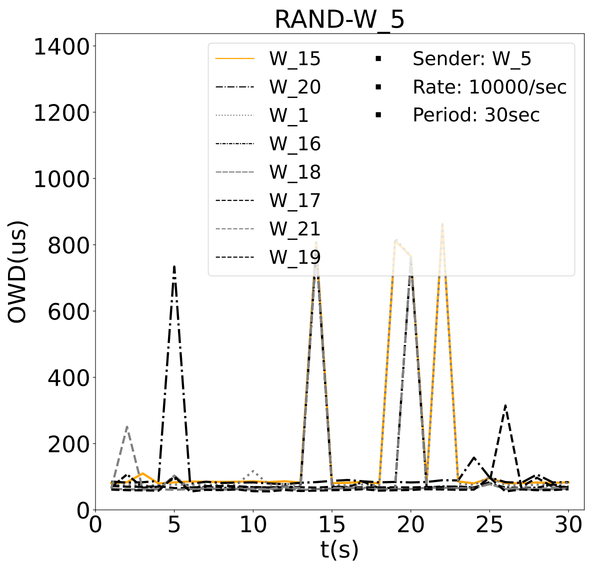


Figure 5.8: Standard-Deviation TSP

	p90	p50	mean	stddev	receiver
RAND	383 us	256 us	281.05 us	308.29 us	W_15

Figure 5.9: Worst Leaf Receiver Key Statistical Metrics

Figure 5.10: **RANDOM Tree Comparison.**

We now compare the `RAND` tree to the previously shown `BEST-WEIGHTED` and `WORST` systems. All of whom share the same root node as to eliminate its influence from the presented results. The weakest performing leaf nodes in the following trees – `BEST-WEIGHTED`, `RAND` and `WORST` – show at their OWD 90th percentile, respectively, 358 μs, 383 μs, and 498 μs latency values. At the 90th percentile, the `BEST-WEIGHTED` worst receiver's latency shows a 6.52% decrease in latency versus that of the `RAND` tree. Furthermore, it is also shown that the `RAND` tree displays more frequent latency spikes in comparison to the `BEST-WEIGHTED` tree. The `RAND` tree's largest standard deviation value of 863.30 μs is much lower than that of the `WORST` tree at 1337.13 μs, yet stil displays a 22.12% increase in contrast to that of the `BEST-WEIGHTED` tree. Finally, comparing Figures 5.7, 5.1 and 5.2, it is noticed that the OWD CDFs of the `RAND` tree are not as sparsed out as seen in the `WORST` tree, but display less uniform latency reception across its leaf nodes versus the `BEST-WEIGHTED` tree. To reinforce these statements and aid in its visualization, we provide in Figures 5.11 and 5.12 a direct comparison of the receiver CDFs and TSPs of the `BEST-WEIGHTED`, `RAND` and `WORST` trees.

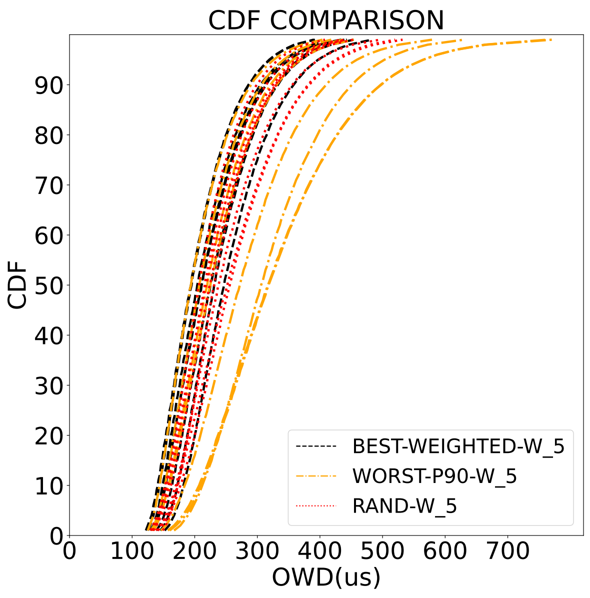


Figure 5.11: Complete OWD CDFs

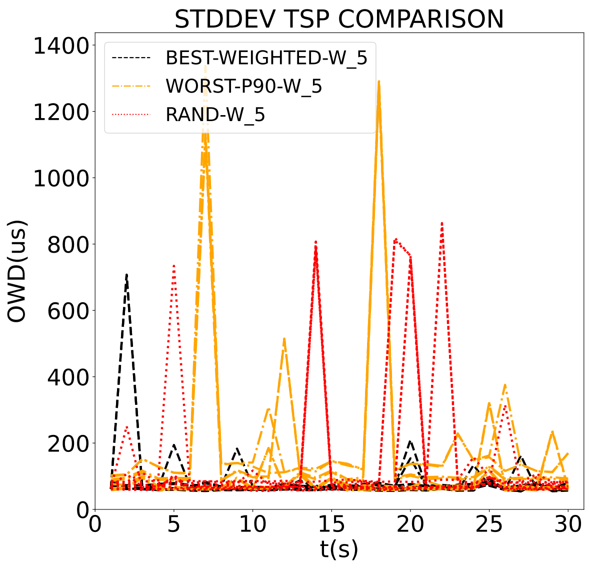


Figure 5.12: Complete TSPs

Figure 5.13: **BEST-WEIGHTED x WORST x RAND comparison.** As expected, randomly selecting nodes for a tree places results within the respective boundaries of the `WORST` and `BEST-WEIGHTED` systems. This serves as a useful baseline to straggler optimization methods and demonstrates their impact versus a vanilla system with no prior selection made.

These results illustrate well the possible improvement that TreeBuilder might have on one's system in contrast to that of a vanilla or a worst-case scenario. Moreover, considering the strict time constraints and network demands of contemporary financial exchanges, even incremental improvements in latency at the microsecond level are crucial. In the following section we move onto other strategies we have employed and compare them amongst each other.

5.2.2 Optimization Strategy Differences

In this section we explain the effect of different construction strategies onto our system's multicast tree. Table 5.5 shows the key OWD statistical metrics of the worst receivers for the `BEST-WEIGHTED`, `BEST-P90`, and `BEST-P50` trees. All of whom shared the same `W_5` selected root. We notice that the `BEST-WEIGHTED` tree's weakest receiver shows a comparable OWD latency at its 90th percentile of 358 μs in comparison to that of the `BEST-P90` tree at 364 μs. However, `BEST-P50` tree's least performant node shows a considerably worst 90th percentile value of 392 μs, which is an 9.50% increase from that of the `BEST-WEIGHTED` tree.

	p90	**p50**	**mean**	**stddev**	**receiver**
BEST-Weighted	358 us	247 us	261.4 us	78.95 us	`W_9`
BEST-P90	364 us	246 us	262.11 us	159.16 us	`W_14`
BEST-P50	392 us	269 us	282.57 us	83.57 us	`W_8`

Table 5.5: Worst Leaf Receiver Key Statistical Metrics per Tree

In Figure 5.14, we present the OWD latency CDF of the worst receiver from each of the above trees. Additionally, Figure 5.15 shows the OWD standard deviation values within each second interval of the evaluation period for those receivers as well.

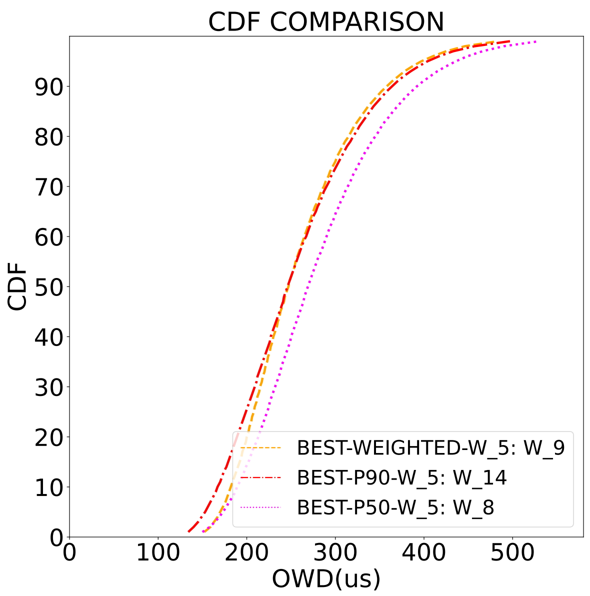


Figure 5.14: CDF per worst receiver

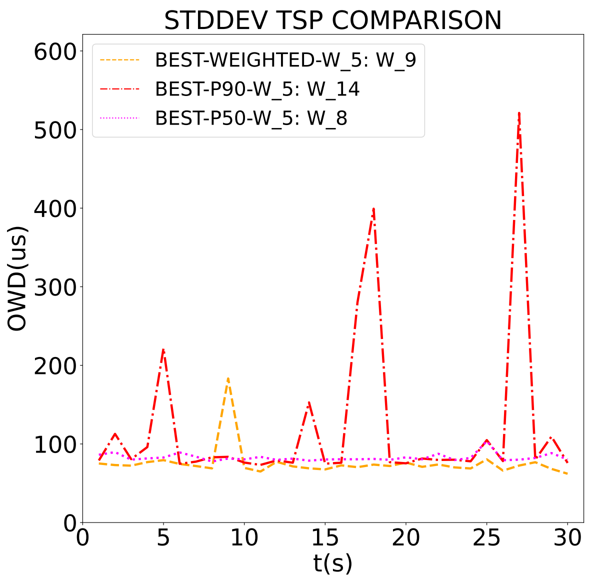


Figure 5.15: TSP per worst receiver

Figure 5.16: **Worst leaves comparison: BEST-WEIGHTED, -P90 and -P50.** The `P50` strategy prioritizes stability over minimizing overall latency. The `P90` strategy aims to avoid high latency, but still selects nodes with significant variability. The `WEIGHTED` method seeks to balance latency and variability, yet is overly sensitive to sporadic spikes.

The P50 strategy selects VMs based on the median, meaning that nodes with more consistent performance, even if not necessarily the lowest latencies, are more likely to be chosen. This results in a system that exhibits less extreme fluctuations. This can be seen in Figure 5.15, as the largest standard deviation value of 521 μs from the BEST-P90 tree's worst receiver is 403.36 % higher than that of the BEST-P50 at 103.53 μs. Moreover, the BEST-P50 tree's receiver is shown to be more robust against variance overall, exhibiting its largest spike to be 43.51 % lower than that of the BEST-WEIGHTED tree at 183.28 μs. Nonetheless, the median alone does not fundamentally optimize for overall reduced latency, as it rather selects stable nodes even if their latency values are not as minimal. This is shown in Figure 5.14, where the BEST-P50 tree's worst node's CDF is, for the most part, right-shifted compared to those of the BEST-WEIGHTED and BEST-P90 trees. Now, we look at the system as a whole as seen by Figures 5.17 and 5.18. These illustrate the OWD CDFs and standard deviation TSPs across all receivers in the three variations of the BEST trees. Furthermore, the individual plots of the BEST-P90 and BEST-P50 trees can be seen for reference in the sections A.5.1 and A.5.2 of the appendix.

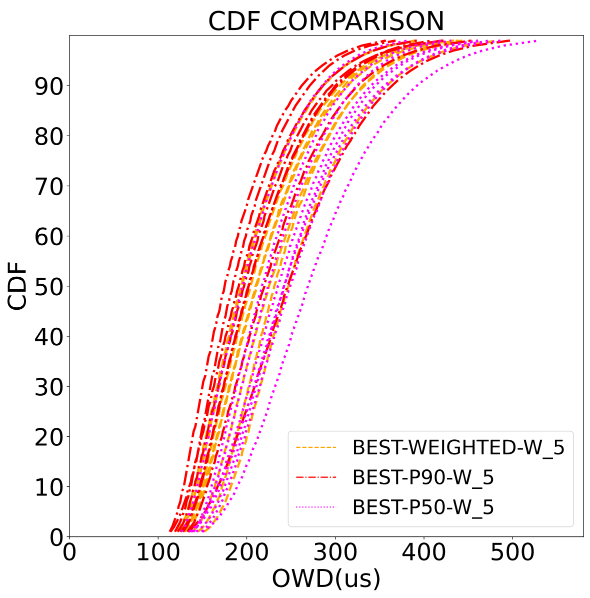


Figure 5.17: Complete OWD CDF

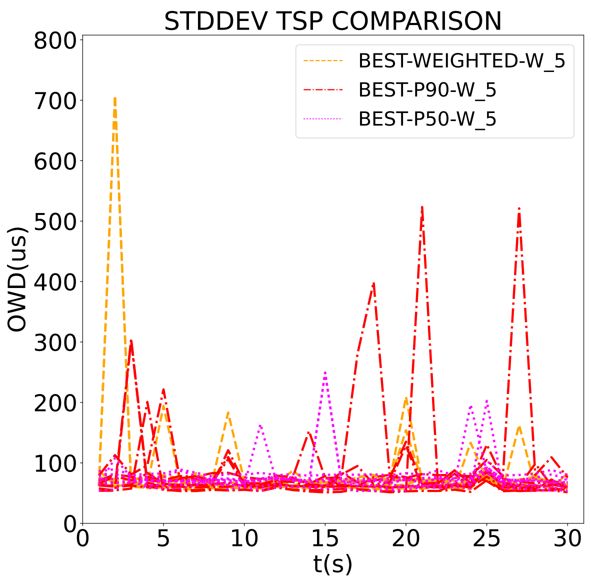


Figure 5.18: Complete OWD TSP

Figure 5.19: **Complete comparison: BEST-WEIGHTED, -P90 and -P50.**

Systematically, the P50 strategy successfully prioritizes stability in latency, whilst BEST-P90 optimizes for better performance especially in higher percentiles. The WEIGHTED method aims to strike a balance and attempts to prune volatile nodes during its selection.

This systematic overview gives us a broader understanding on how these different strategies affect their final multicast tree. The P90 method, as it ranks VMs by their 90th percentile OWD latency, is seeking to pick nodes that are more performant in higher percentiles. Yet, it might still include VMs with significant variability. By utilizing the median, as a robust measure of central tendency, the P50 method effectively ignores the impact of extreme outliers, filtering out occasional latency spikes during evaluation. Furthermore, the P50 strategy inherently includes some sensitivity to frequent fluctuations, as if often enough will affect the system's median. This can be seen by the low standard deviation values throughout BEST-P50's tree in Figure 5.18. Finally, it is suggested that the P90 and WEIGHTED strategies are better at selecting VMs that provide lower latency overall, as they eliminate machines with high outliers. However, as observed from the standard deviation time series (Figure 5.18), the BEST-WEIGHTED tree exhibits a more predictable latency behavior, with less frequent spikes than that of the BEST-P90 tree. Since the WEIGHTED strategy includes the OWD latency standard deviation of a machine during its evaluation, it is able to filter out volatile VMs. Conversely, given this aggressive weighting of standard deviation, the WEIGHTED method can preemptively eliminate VMs based on sporadic latency spikes. This might inadvertently favor VMs that have poorer performance or that display consistently variant behavior, yet lower than that of the given spike. As a result, the emphasis on minimizing variance can overshadow the selection of VMs that suffered occasional anomalies.

5.2.3 Root Selection Effect

In this section, we explore the effect of root selections on the overall system. Figures 5.20 and 5.21 show the OWD latency CDFs and the standard deviation TSPs for all receivers in the BEST-WEIGHTED trees. Each BEST-WEIGHTED tree fixed its root node as W_5 and W_3, respectively, before constructing its system.

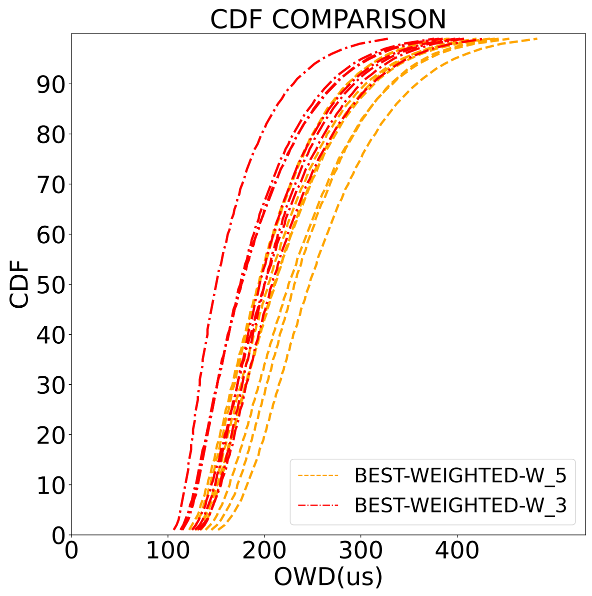


Figure 5.20: Complete OWD CDF

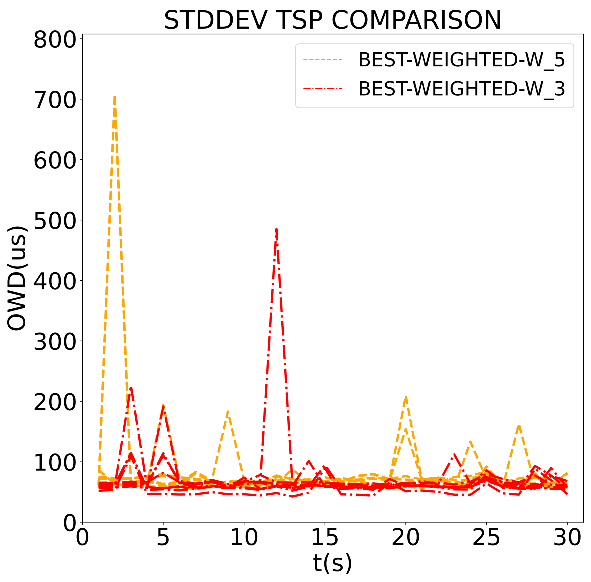


Figure 5.21: Complete OWD TSP

Figure 5.22: **BEST-Weighted W_5 Vs. W_3 Root Comparison.** The overall receivers of the W_5 root-based tree are systematically under-performing comparative to their counterpart based on W_3.

Root selection is done as the very first step to our method. Additionally, we have fixed the seed to the random generator utilized by the manager node, as to ensure that it serves the same first **K** nodes to every tree. Even though this is not a direct comparison of the same tree with a substituted root, the choice of a root node does affect which children are appended as the tree's first layer and therefore influences its descendants.

We can observe from Figure 5.20 that in the case of the `BEST-WEIGHTED` trees there is a systematic difference to their receivers. Receiver nodes of the `W_5`-root tree, almost as a whole, are right-shifted in comparison to their `W_3` counterpart. That same phenomenon is seen in Figure 5.26 for the `BEST-P50` trees. Most importantly, it follows the same pattern where the least performant system is based on `W_5`. Furthermore, in `BEST-P50's` case, the systematic right shift of CDFs is so expressive that their lines are almost completely separated. However, for both `BEST-P50` and `BEST-WEIGHTED` trees, the standard deviation values do not exhibit any clear pattern.

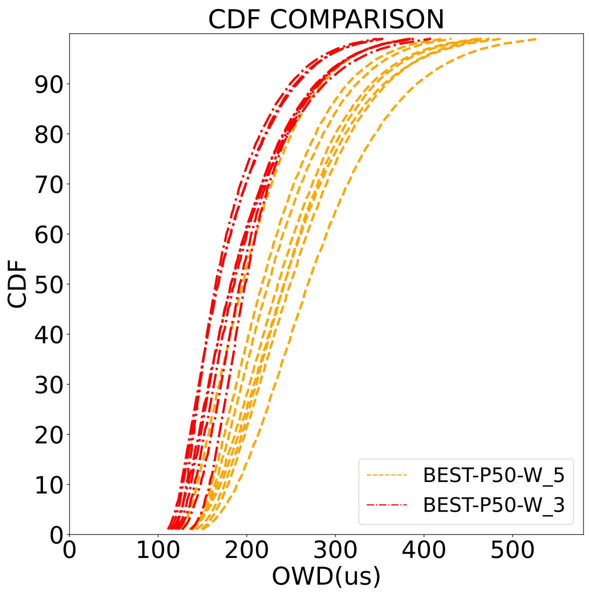


Figure 5.23: Complete OWD CDF

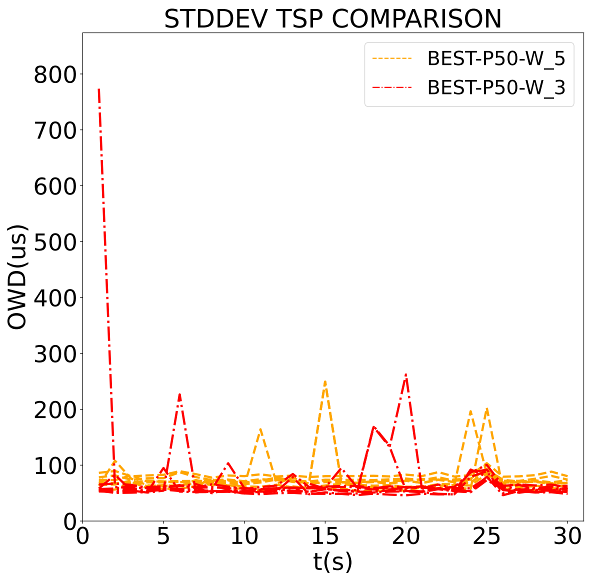


Figure 5.24: Complete OWD TSP

Figure 5.25: **BEST-P50 W_5 Vs. W_3 Root Comparison.** The difference in the both systems are so expressive that the `W_5` root based tree within the same selection strategy almost has all receivers cleanly shifted to the right.

Finally, looking at Figures 5.26 and 5.27, it is clear that the worst receivers of the `BEST-P90` trees are in fact belonging to the `W_5` based system. However, in this scenario the separation or right-shift of the `W_5` based system is not nearly as strong as in the previous cases. Statements regarding the standard deviation values are also inconclusive.

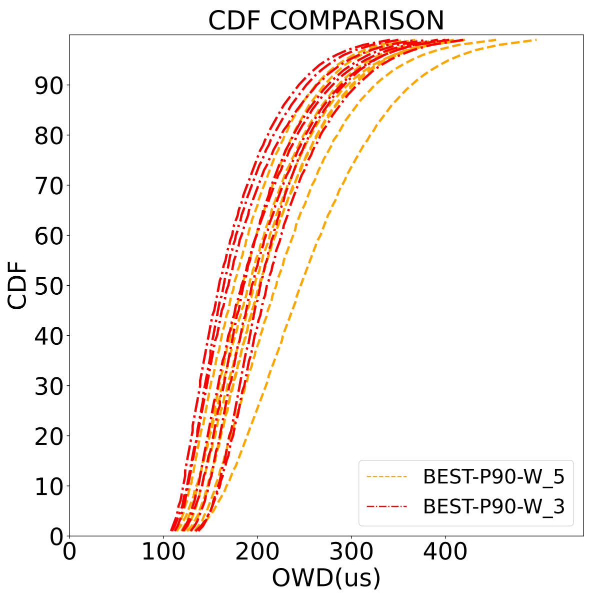


Figure 5.26: Complete OWD CDF

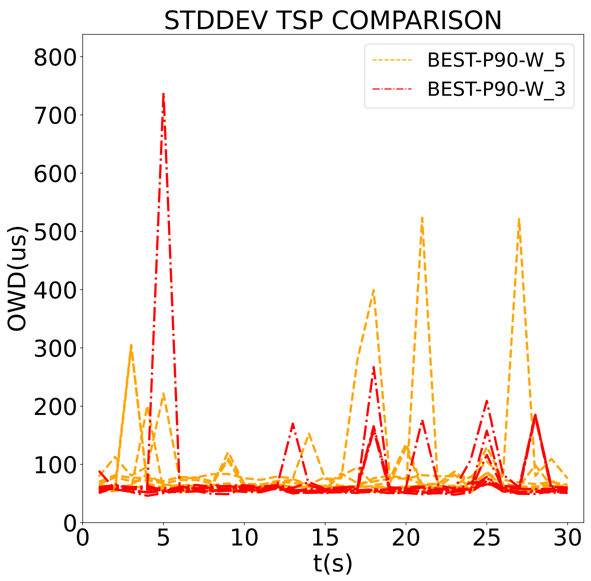


Figure 5.27: Complete OWD TSP

Figure 5.28: **BEST-P90 W_5 Vs. W_3 Root Comparison.** The worst receivers are also from the W_5 tree, but systematically the CDF shifts are less expressive than the BEST-P90 and BEST-WEIGHTED trees.

5.2.4 LemonDrop Comparison

We now delve into the question of how does TreeBuilder compare against LemonDrop [Sac22]. A separate experiment was conducted with the same parameters as the one currently exposed, however focusing solely on LemonDrop's method and varying its hyperparameter ϵ and the routine's maximum number of iterations (algorithm 1). Their results can be seen in appendix A.5.6. From that analysis we have observed that there were improvements to the system by decreasing ϵ, as to allow for the routine to be more selective on solution convergence. However, these benefits were outweighed by the long convergence times in the seconds range, which do not fit into the scope of cloud-based financial exchanges. Therefore, we have fixed LemonDrop's ϵ for our currently shown experiment as 1×10^{-4} and its maximum number of iterations to 1000. Table 5.6 shows the summarized metrics to the worst receivers results across the BEST-WEIGHTED and LEMON trees.

	p90	**p50**	**mean**	**stddev**	**worker**
BEST-Weighted: W_5	358 us	247 us	261.4 us	78.95 us	W_9
BEST-Weighted: W_3	312 us	208 us	223.1 us	68.6 us	W_9
LEMON-1-W_4	338 us	236 us	249.5 us	94.94 us	W_9
LEMON-2-W_5	325 us	223 us	236.7 us	76.39 us	W_9

Table 5.6: **Worst Leaf Receiver Key Statistical Metrics per Tree.** Comparable results regarding the worst receivers of the two systems.

It is important to note that the `LEMON-1-W_4` and `LEMON-2-W_5` trees are each a separate run of our implemented LemonDrop routine as seen in appendix A.4.1. They also have separately done data collection phases. Furthermore, we also include both `BEST-WEIGHTED` trees across both differently selected roots. Table 5.6 shows us comparable results regarding the worst receiver of each tree, especially in the 90th percentile. Additionally, we compare the individual CDFs of these receivers in Figure 5.29. Interestingly enough, the worst leaf node in all of the 4 systems is the same `W_9` node. However, it does display a different behavior as their tree configurations are distinct. We observe that TreeBuilder's `W_3` based `BEST-WEIGHTED` tree exhibited a healthier OWD latency profile of from its worst receiver than the rest. Furthermore, it is seen that the `W_3` based `BEST-WEIGHTED` tree's least performant node exhibits the most stable behavior regarding latency spikes over the experiment duration. This is shown by Figure 5.30.

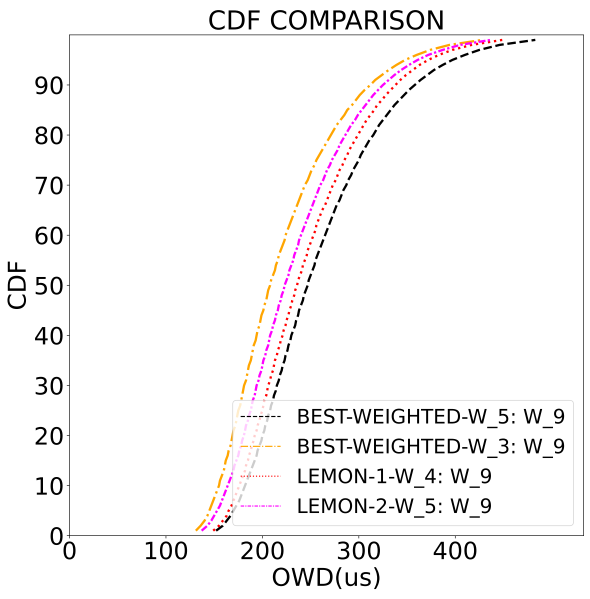


Figure 5.29: Worst receiver CDF

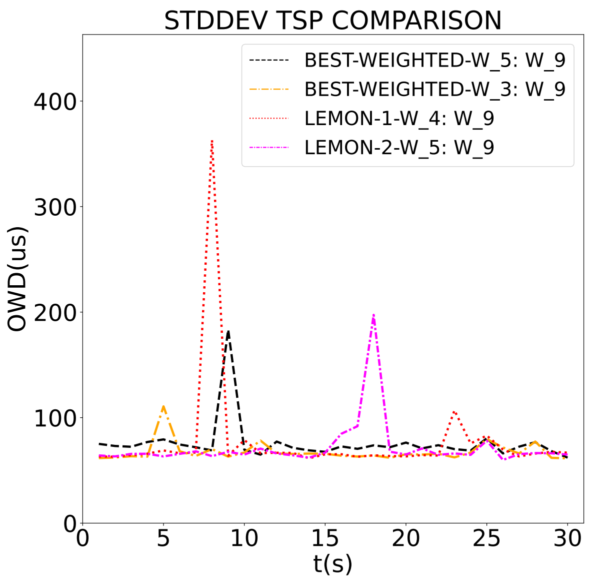


Figure 5.30: TSP per worst receiver

Figure 5.31: **Worst leaves comparison: BEST-WEIGHTED, LEMON.** `LEMON`-based systems seem to more robust against variance in the cloud. However, regarding the worst receiver node, `BEST-WEIGHTED` was able to achieve a more performant CDF in its `W_3` based variation.

Now, with Figures 5.32 and 5.33, we look at all receivers of the mentioned trees. We notice that on a systematic level, the `LEMON` trees are more robust in terms of latency variance. This is a natural conclusion given the use of median OWD latency values in LemonDrop's OWD Matrix. Moreover, this has also been observed in our comparisons of the `BEST-P50` and `BEST-WEIGHTED` trees in section 5.2.2.

This couples well with Figure 5.32, as we observe the CDFs of all receivers directly contrasted per tree system. Given that LemonDrop is not allowed a great deal of time to converge to a highly optimal solution due to the selected ϵ and max iteration values, it might not be able to search the solution space long enough to provide an unequivocally better system. However, in the case of the `LEMON-1-W_5` tree, its better performing node is quite close to that of the `BEST-WEIGHTED-W_3` node. Additionally, the `BEST-WEIGHTED-W_5` tree seems to be systematically worse than that of the `LEMON` trees, which might have to do with its chosen root. Finally, it is seen that the `BEST-WEIGHTED-W_3` tree might have as a majority better performing nodes than those of the `LEMON` types. Given `WEIGHTED's` strategy based on the 90th percentile and standard deviation of the OWD values of a machine, this might push the `BEST-WEIGHTED` tree to look for more performant VMs. However, due to its sensitivity to spikes, this method may also lead to the elimination of suitable instances. This could also explain the underperformance observed in the `BEST-WEIGHTED-W_5` tree in comparison to that of the `LEMON` trees. This tells us that the `WEIGHTED` formula is still inconsistent and does not capture in entirety the essence of a node's ingress performance to make strong choices.

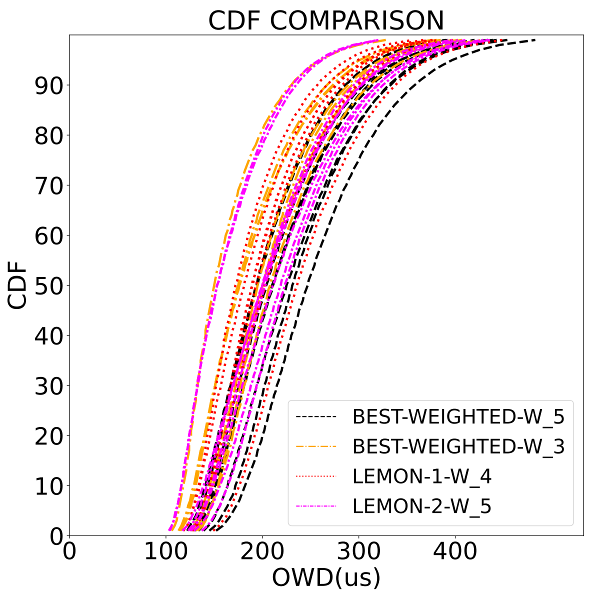


Figure 5.32: Complete OWD CDFs

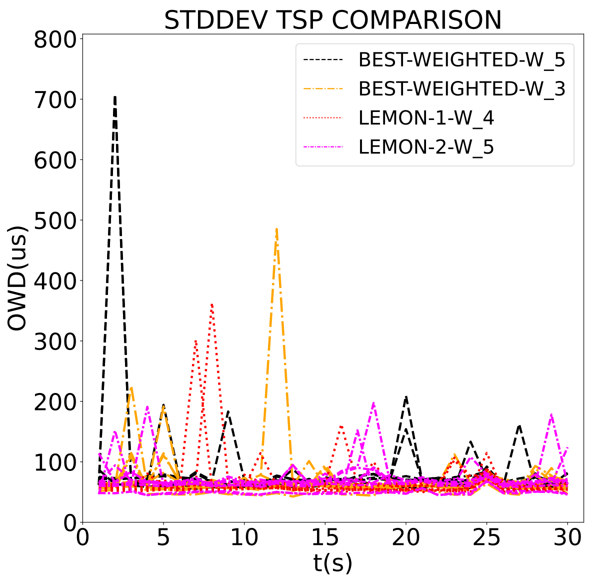


Figure 5.33: Complete TSPs

Figure 5.34: **Complete comparison: BEST-WEIGHTED, LEMON.**

Chapter 6

Conclusions and Outlook

In this project, we developed TreeBuilder, a tool that analyzes VMs in a cluster and incrementally forms a multicast tree of a given depth and fan-out. We introduced a generic and straight-forward strategy-based mechanism to select machines within a pool's subset, which allows for extensive exploration of different selection algorithms. The strategies we implemented – `WEIGHTED`, `P90`, and `P50` – have already demonstrated considerable effects on the final system's behavior.

Furthermore, we provided a faithful implementation of LemonDrop, which is a modern VM selection and scheduling heuristic optimized for an application's latency needs. We demonstrated that, given a tree topology and the context of cloud-based financial exchanges, TreeBuilder performs well comparibly to LemonDrop's built system. An instance of TreeBuilder's `BEST-WEIGHTED` tree exhibited the lowest 90th percentile latency for its worst receiver at 312 µs, approximately 7.7% lower than the worst receiver of LemonDrop's tree (`LEMON`), which had a latency of 338 µs. Similarly, the `BEST-WEIGHTED` tree's worst receiver's mean latency was 223.1 µs, around 10.6% lower than that of `LEMON` at 249.5 µs. However, LemonDrop's tree exhibited lower standard deviation values throughout the experiment, indicating a more stable system. We also compare TreeBuilder's results against a vanilla baseline (`RAND`), where no prior selection of VMs in a cluster are made, and a worst-case system `WORST`, where nodes are selected in favor of underperformance. At the 90th percentile, the `BEST-WEIGHTED` worst receiver's latency shows a 6.52% decrease in latency versus that of the `RAND` tree. and a 28.11% decrease than that of the `WORST` tree. Considering the strict network demands of modern financial exchanges, even incremental improvements in latency at the microsecond level are crucial.

Nonetheless, there are valuable insights to be drawn from the trade-offs observed between TreeBuilder's different strategies and LemonDrop's optimization framework. LemonDrop's and TreeBuilder's median-based selection strategies showed more robust systems in addressing cloud latency variance. We believe that future work on refining TreeBuilder's heuristic for VM selection is necessary, as there is still value in optimizing for high performance while maintaining latency predictability. Apart from that, we observed that a root selection method could further improve TreeBuilder, as demonstrated by the results of trees with identical strategies but distinct root nodes. Finally, given TreeBuilder's generic design, it can be further extended to analyze distributed architectures beyond multicast trees, offering an extensible solution for cloud-based multicasting with significant potential for further enhancement.

Appendix A

Artifacts

The repository for this work is available on Github under:

- https://github.com/duclos-cavalcanti/TreeBuilder

```
[duclos@localhost TreeBuilder] $ ls
analysis/   docs/       manager/    tools/          notebook.ipynb
bin/        examples/   schemas/    CMakeLists.txt
build/      infra/      scripts/    Makefile
cmake/      jasper/     src/        README
deploy/     lib/        test/       LICENSE
```

Listing A.1: Repository File-Structure

The project is organized, for the most part, between python modules and cmake-based C++ executables. Python modules are callable via `python3 -m <module>` and serve as a convenient way to compartimentalize logical units. These were placed in the `analysis`, `manager`, and `deploy` directories. All Terraform and Packer `hcl` modules are placed under `infra`. At the root of this repository lies a `CMakeLists.txt` file used to compile the needed executables, who are to be found in the `bin` directory. Within `schemas` is where the JSON files used to conduct experiment parameters are placed, as one calls the `deploy` module with its needed arguments. Third-party libraries such as ZeroMQ and nlohmann's `json` are built in `lib` and are ignored by git [Hc23, Loh24]. The `analysis` module is used to both download data from a given cloud provider's storage and process an experiment's results. A jupyter-notebook is also available for interactive plotting. Unit-tests used to troubleshoot TreeBuilder features and individual objects can be seen in `test`, leveraging the `pytest` library [K+24]. Finally, a `Makefile` is found at the repository's root directory for frequent command-line instruction calls.

A.1 Building

Steps needed to compile TreeBuilder's job executables, which are then found in `bin`.

```
cd build
cmake .. && make
```

Listing A.2: Building with CMake

A.2 Execution

Manager Module

```
python3 -m manager -a [manager,worker] -n <name> \
        -i <ip_address> -p <port> -s <path_to_schema_file>
```

Listing A.3: Manager Python Module

Deploy Module

```
python3 -m deploy -a [plan, destroy, build, deploy] \
        -m [default, test, udp, mcast, lemon, lemondrop]  \
        -i [gcp, docker] -f <fanout> -d <depth> -r <rate> \
        -t <duration> -e <evaluation> -s <size> -n <number> \
        -c <root_choices> -w <warmup>
```

Listing A.4: Deploy Python Module

Analysis Module

```
python3 -m analysis -a [pull,process] -i [gcp, docker] -p <prefix>
```

Listing A.5: Analysis Python Module

Parent, Child, Mcast and Lemon CLI Arguments

```
./parent -a [addr_1 addr_2 ... ] -r <rate> -d <dur> -w <warmup>
./child -i <ipaddr> -p <port> -r <rate> -d <dur> -w <warmup>
```

Listing A.6: parent and child CLI Command

```
./mcast -a [addr_1 addr_2 ...] -r <rate> -d <duration> \
        -i ipaddr -p port -w <warmup> -n <name> -R
```

Listing A.7: mcast CLI Command

```
./lemon -i <ipaddr> -p <port> -w <warmup> -f <start> -s <schema>
```

Listing A.8: lemon CLI Command

A.3 Data Structures

A.3.1 Tree

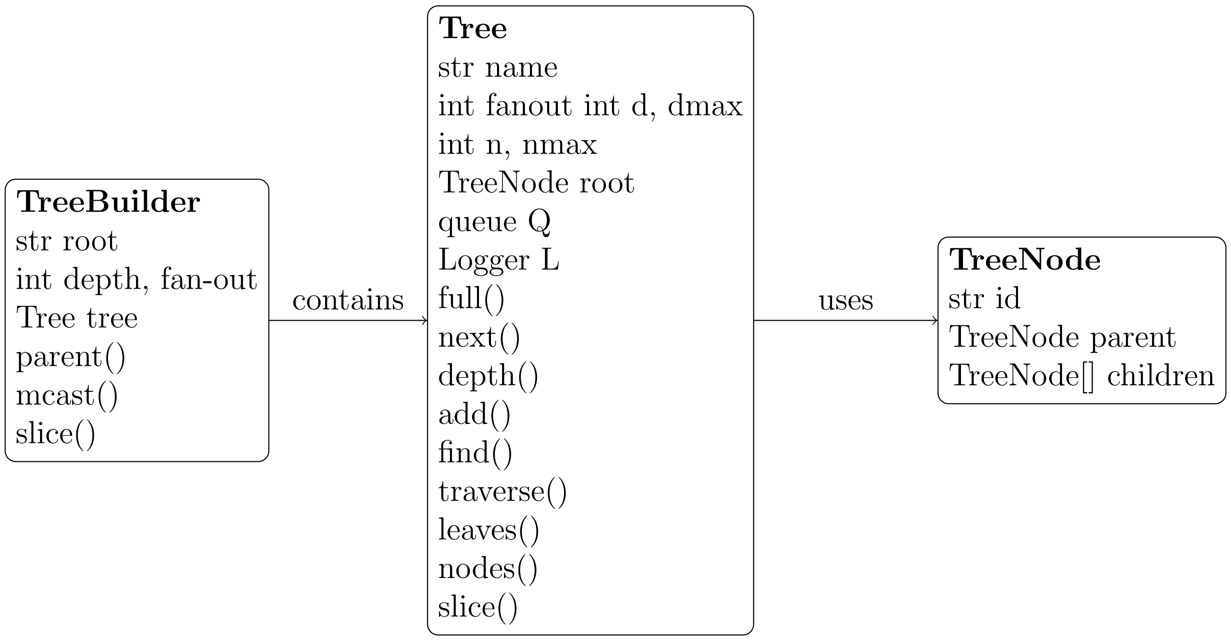


Figure A.1: Tree Data Structures

A.3.2 Pool, Seed, Logger and Timer

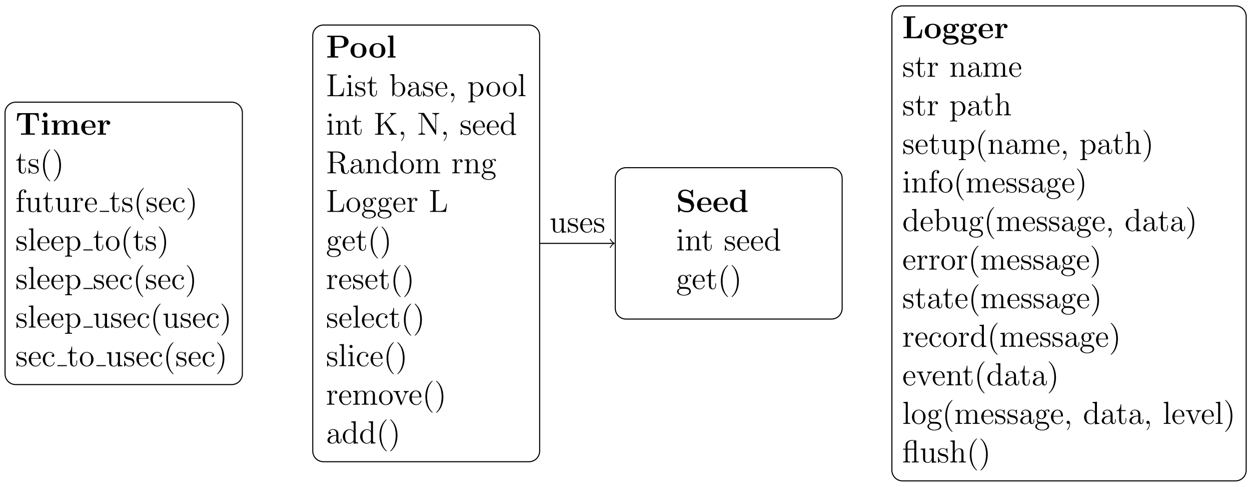


Figure A.2: Pool, Seed, Timer and Logger Data Structures

A.4 Implementation

A.4.1 FAQ

```python
    import numpy as np
    from scipy import optimize
    def FAQ(OWD:np.ndarray, LOAD:np.ndarray, N, epsilon, max_i):
        # P(0) = 11T, a doubly stochastic matrix
        P = np.ones((N, N)) / N
        i = 0; s = 0; converged = False
        while s == 0:
            # GRADIENT WITH RESPECT TO CURRENT SOLUTION
            grad = - (LOAD @ P @ OWD.T) - (LOAD.T @ P @ OWD)
            # DIRECTION WHICH MINIMIZES 1ST ORDER APPROX OF f(P)
            Q = np.zeros((N, N))
            row, col = optimize.linear_sum_assignment(-grad)
            Q[row, col] = 1
            # BEST STEP SIZE IN CHOSEN DIRECTION
            func   = lambda alpha: np.trace(
            LOAD @ (P + alpha * Q) @ OWD.T @ (P + alpha * Q).T
            )
            result = optimize.minimize_scalar(
                    func,
                    bounds=(0, 1),
                    method='bounded'
            )
            alpha = result.x
            # UPDATE OUR CURRENT SOLUTION
            P_NEXT  = P + (alpha * Q)
            # STOPPING CONDITIONS
            if np.linalg.norm(P_NEXT - P, ord='fro') < epsilon:
                converged = True
                s = 1
            if i >= max_i:
                s = 1

            P = P_NEXT
            i += 1
        # PROJECTION OF P ONTO P[FINAL]
        P_FINAL = np.zeros((N, N))
        row, col = optimize.linear_sum_assignment(-P)
        P_FINAL[row, col] = 1

        return P_FINAL, converged  # Tuple[np.ndarray, bool]
```

Listing A.9: LemonDrop's adapted FAQ in Python

A.4.2 Experiment Control

```python
class Run():
    def __init__(self, run:RunDict, root, nodes, seed):
        self.data:RunDict = {
                "name": run["name"],
                "strategy": StrategyDict(run["strategy"]),
                "parameters": ParametersDict(run["parameters"]),
                "tree": TreeDict(run["tree"]),
                "pool": [n for n in nodes],
                "stages": [],
                "perf": [ResultDict(p) for p in run["perf"]],
                "timers": TimersDict({})
        }
        self.pool = Pool([n for n in nodes],
                         run["parameters"]["hyperparameter"],
                         seed)
        self.tree  = Tree(name=run["name"], root=root,
                          fanout=run["tree"]["fanout"],
                          depth=run["tree"]["depth"])

class Experiment():
    def __init__(self, schema:dict):
        self.manager = f"{schema['addrs'][0]}"
        self.workers = [ f"{addr}:{schema['port']}"
                         for addr in schema["addrs"][1:] ]
        self.seed    = Seed()
        self.map     = schema["map"]
        self.runs    = []
        pairs        = []
        choices      = schema["runs"][0]["parameters"]["choices"]
        for _ in range(choices):
            pool    = Pool(self.workers, 0, self.seed.get())
            root    = pool.select()
            workers = pool.get()
            pairs.append((root, workers))

        for run in schema["runs"]:
            name = run["name"]
            for i, (root, workers) in enumerate(pairs):
                self.runs.append(Run(run["name"],
                                     root,
                                     self.workers,
                                     self.seed.get()) )
```

Listing A.10: Experiment parsing and control process

A.5 Results

A.5.1 BEST-P90

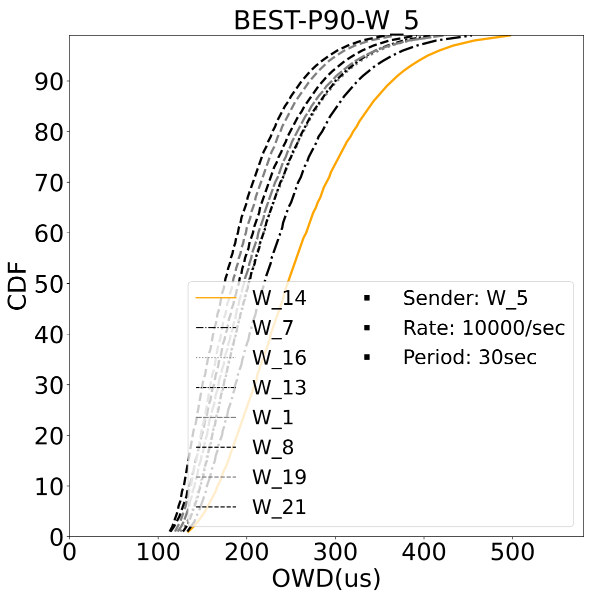


Figure A.3: OWD Latency CDF

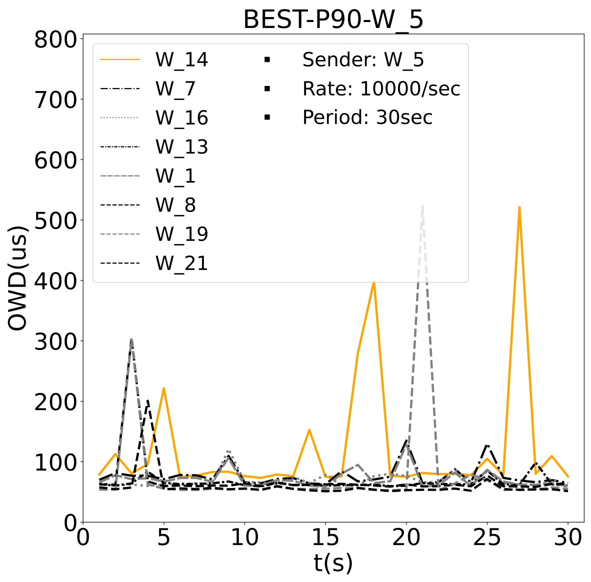


Figure A.4: Standard-Deviation TSP

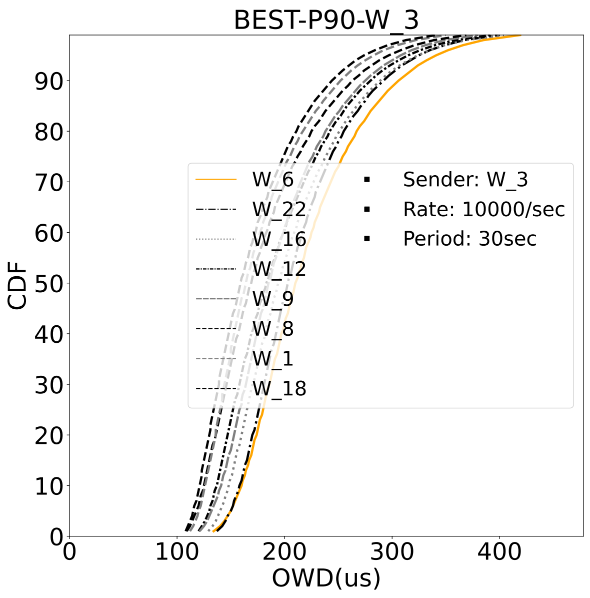


Figure A.5: OWD Latency CDF

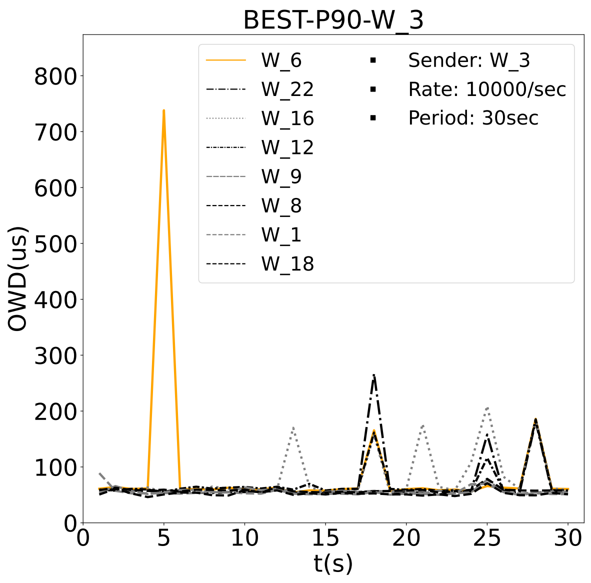


Figure A.6: Standard-Deviation TSP

A.5.2 BEST-P50

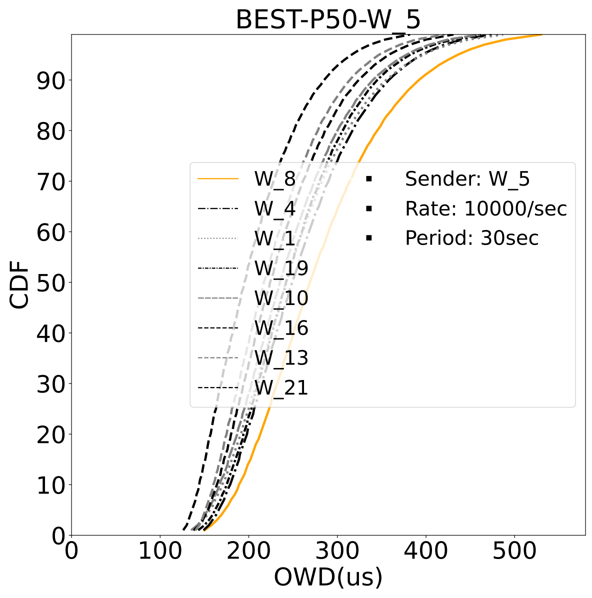


Figure A.7: OWD Latency CDF

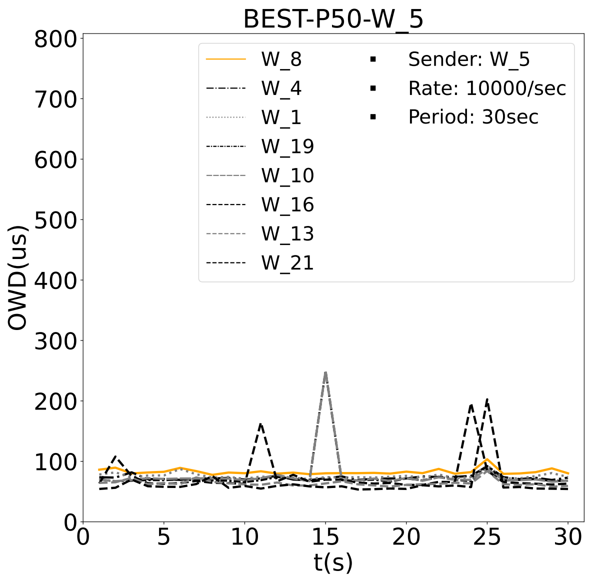


Figure A.8: Standard-Deviation TSP

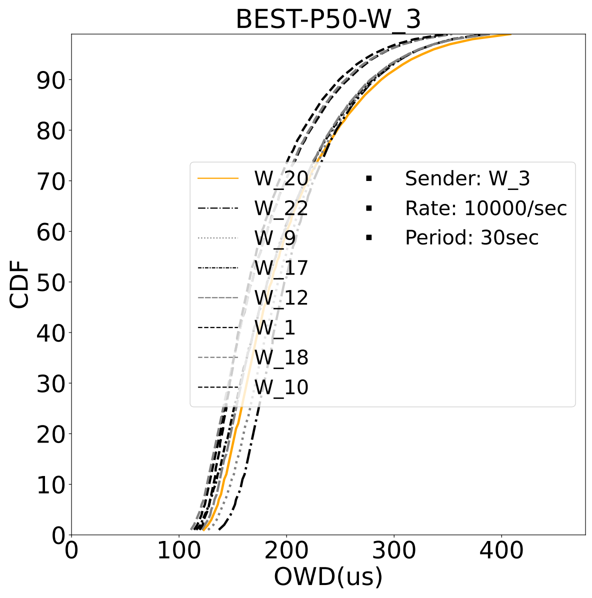


Figure A.9: OWD Latency CDF

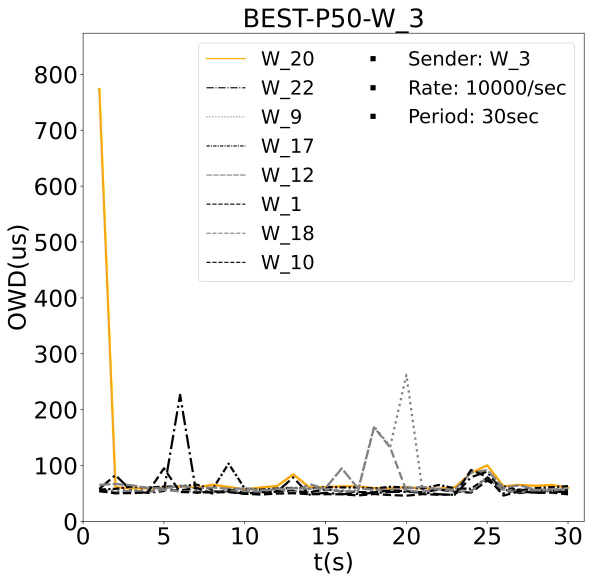


Figure A.10: Standard-Deviation TSP

A.5.3 BEST-WEIGHTED

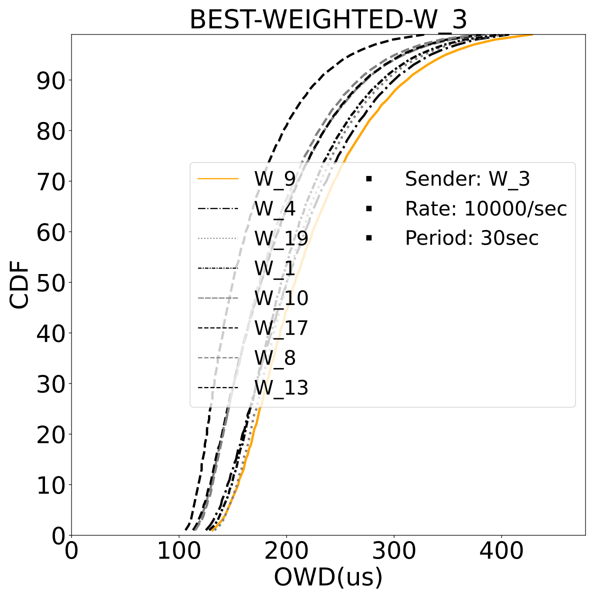


Figure A.11: OWD Latency CDF

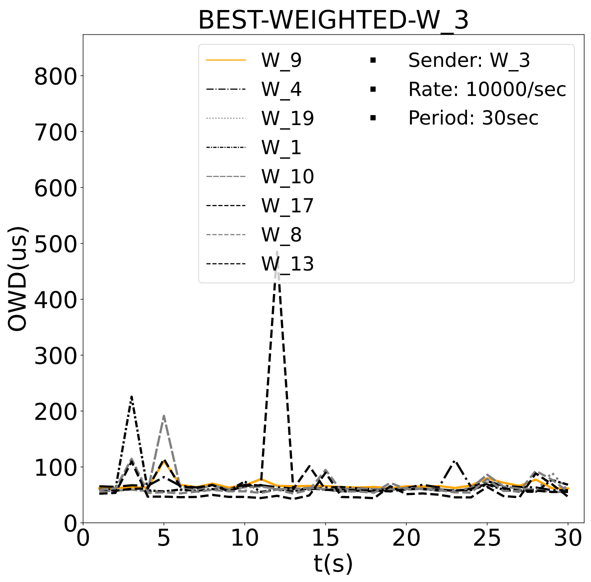


Figure A.12: Standard Deviation TSP

A.5.4 WORST-P90

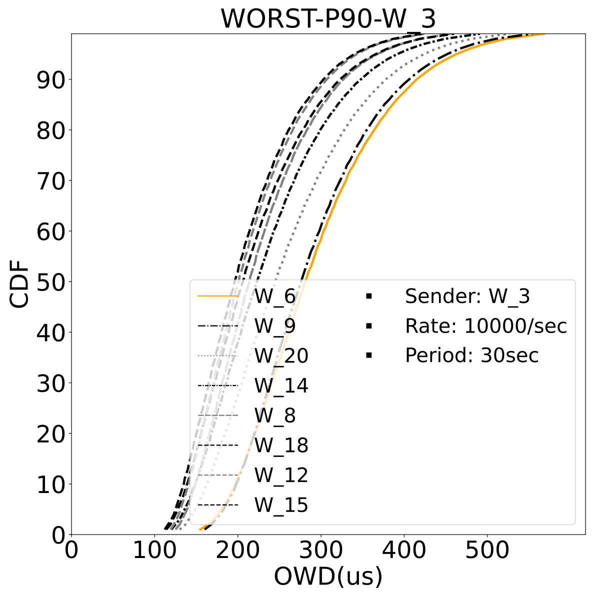


Figure A.13: OWD Latency CDF

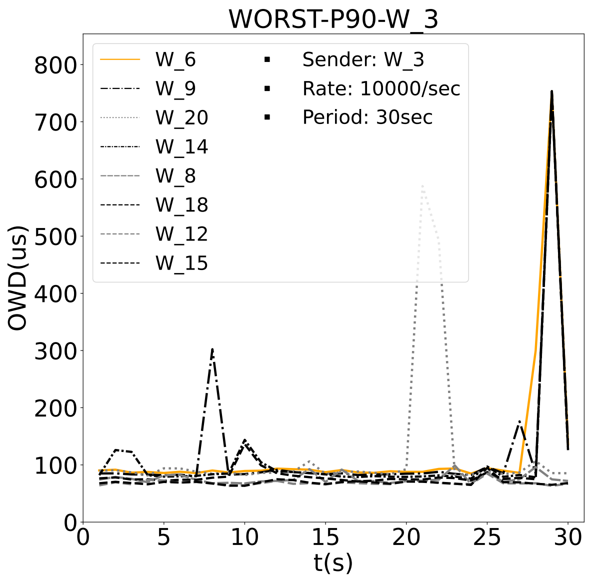


Figure A.14: Standard Deviation TSP

A.5.5 LEMON

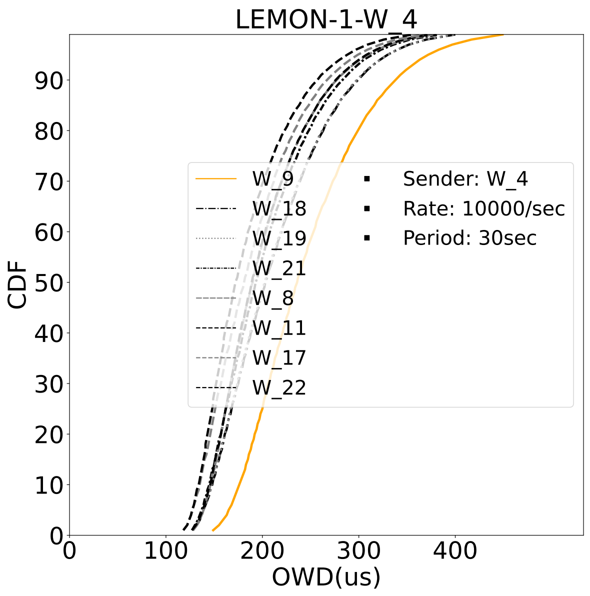


Figure A.15: OWD Latency CDF

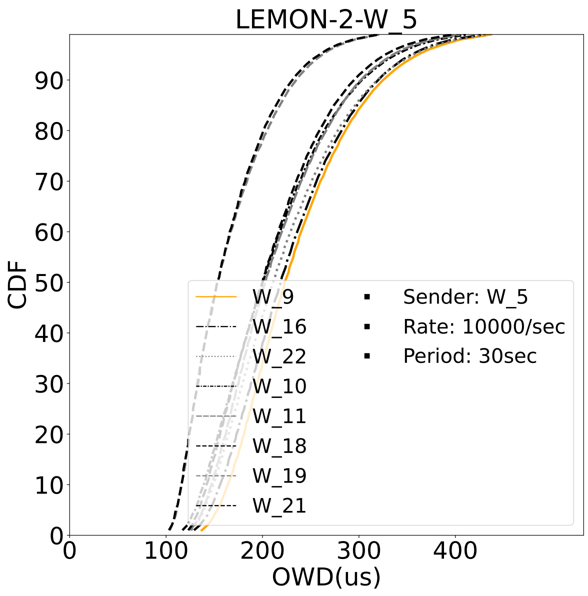


Figure A.16: OWD Latency CDF

Figure A.17: **LEMON-1 and LEMON-2 complete CDFs.**

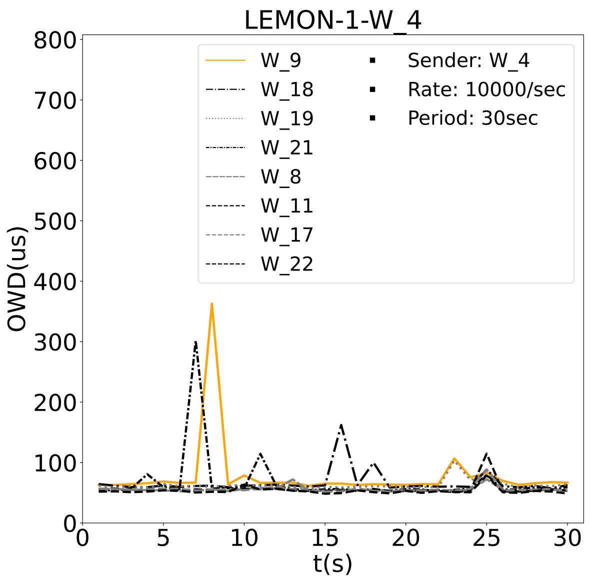


Figure A.18: Standard Deviation TSP

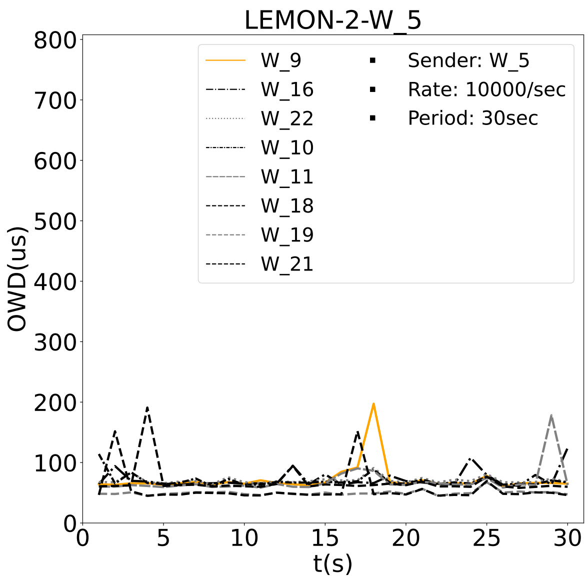


Figure A.19: Standard Deviation TSP

Figure A.20: **LEMON-1 and LEMON-2 complete TSPs.**

A.5.6 LemonDrop Parameter Investigation

OWD Latency CDFs

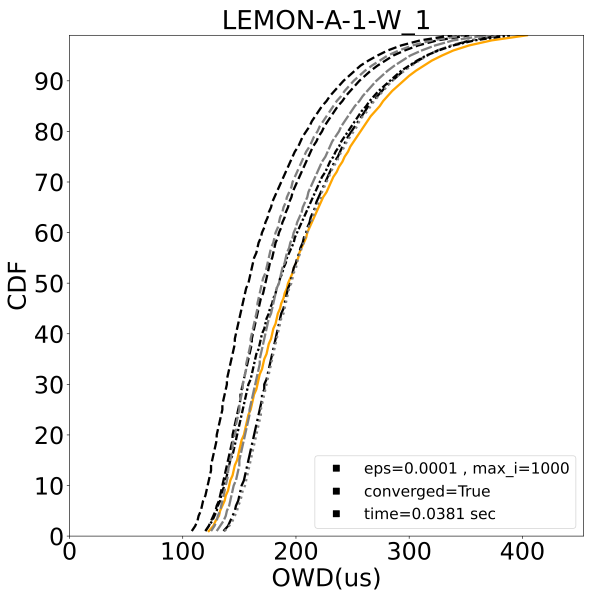


Figure A.21: OWD Latency CDF

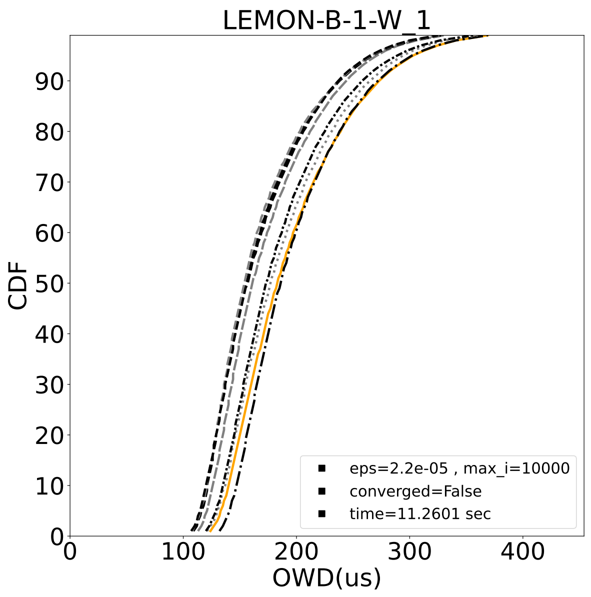


Figure A.22: OWD Latency CDF

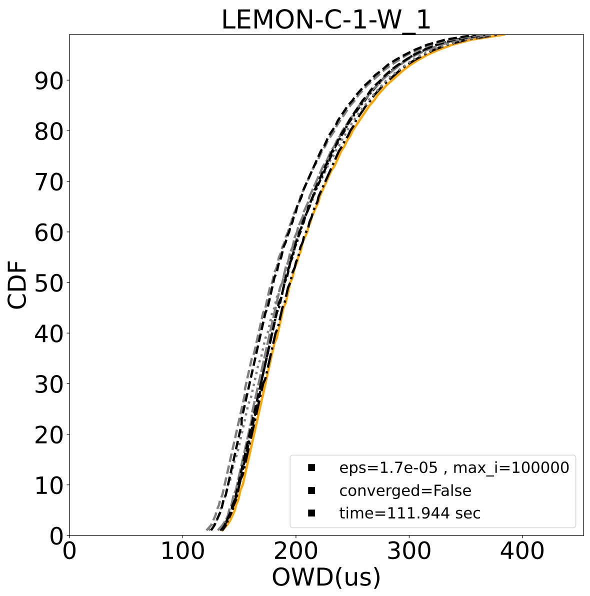


Figure A.23: Direct CDF comparison

	p90	**stddev**
LEMON-A-1: `W_22`	295 us	83.35 us
LEMON-B-1: `W_11`	273 us	64.19 us
LEMON-C-1: `W_17`	285 us	57.64 us

Figure A.24: Worst receiver summary

Figure A.25: **LemonDrop Parametrized CDFs.**

OWD Latency TSPs

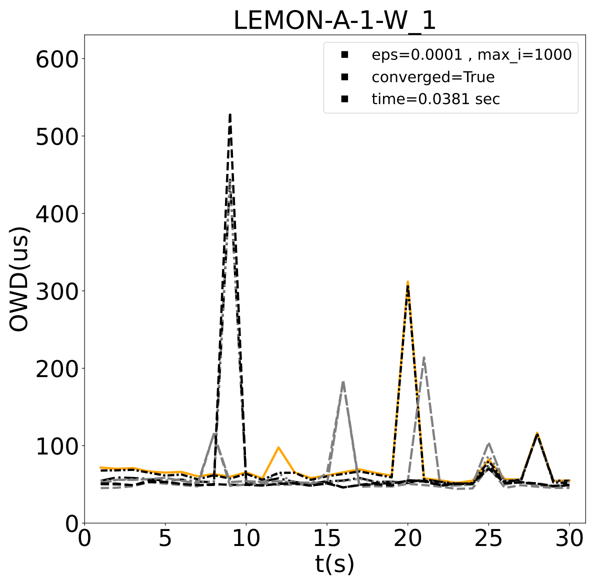


Figure A.26: Standard Deviation TSP

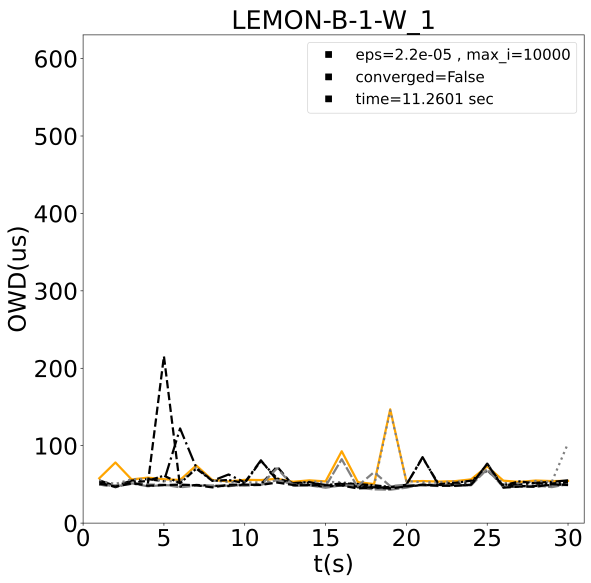


Figure A.27: Standard Deviation TSP

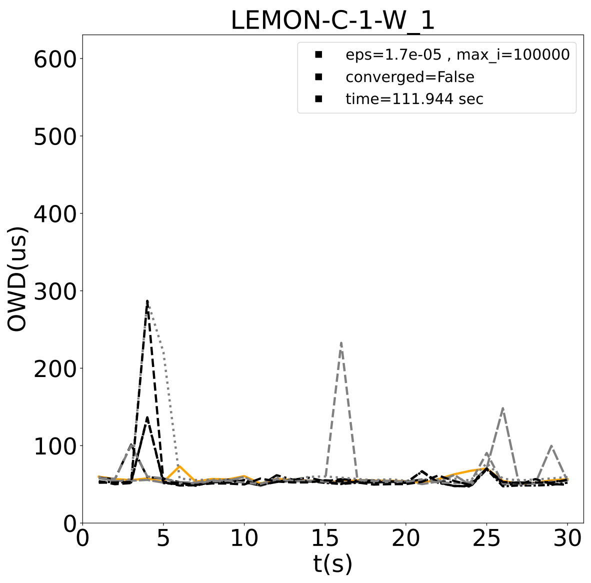


Figure A.28: Standard Deviation TSP

Figure A.29: **LemonDrop Parametrized TSPs.**

Complete Comparison

- `LEMON-A-1`: $\epsilon = 1 \times 10^{-4}$, max_i=1000, time=0.0381sec
- `LEMON-B-1`: $\epsilon = 2.2 \times 10^{-5}$, max_i=10000, time=11.2601sec
- `LEMON-C-1`: $\epsilon = 1.7 \times 10^{-5}$, max_i=100000, time=111.94sec

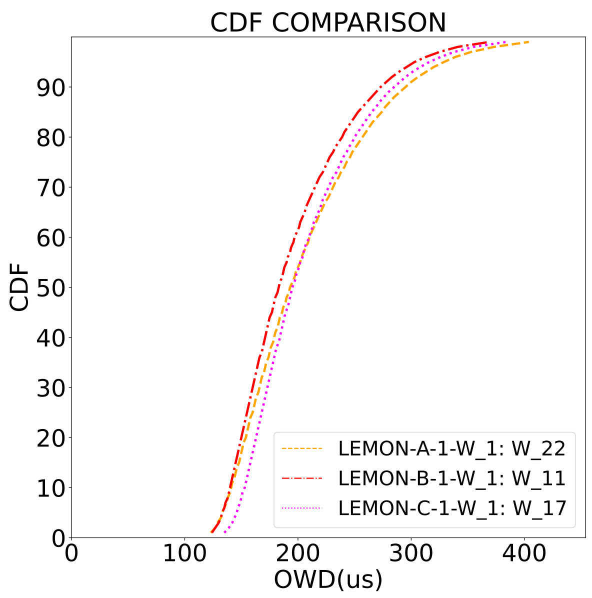


Figure A.30: CDF per worst receiver

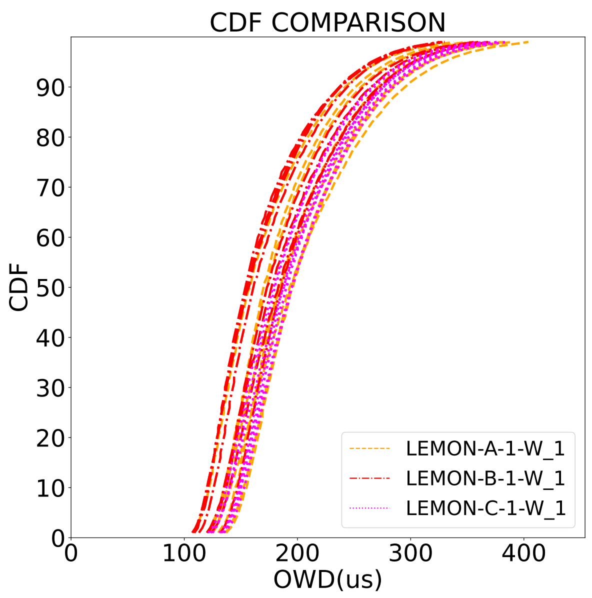


Figure A.31: Complete OWD CDFs

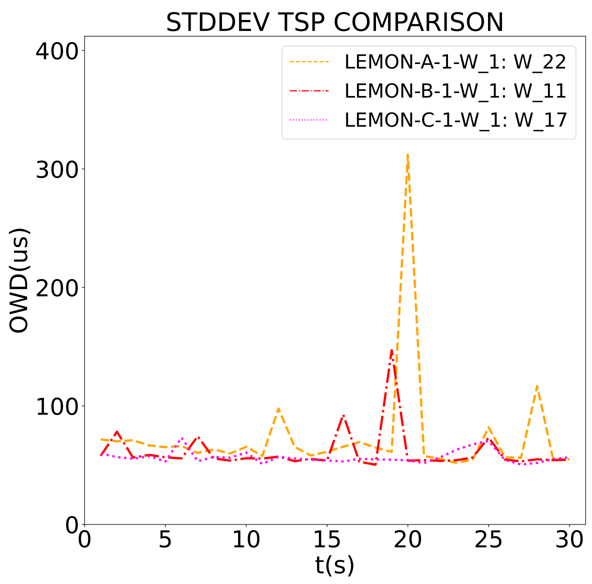


Figure A.32: TSP per worst receiver

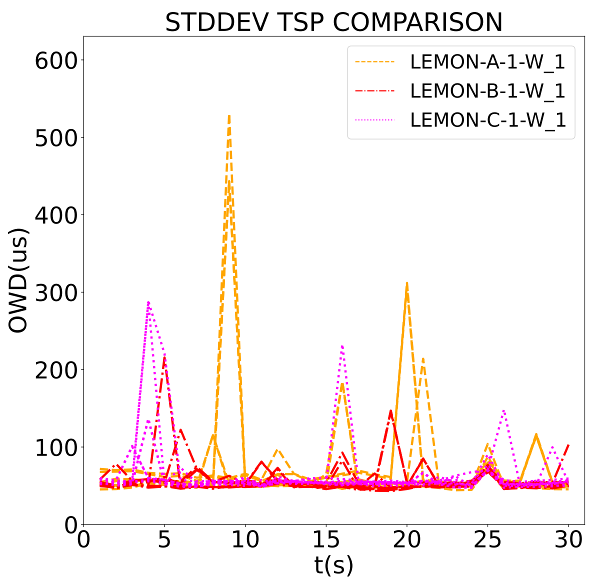


Figure A.33: Complete OWD CDFs

Figure A.34: **LemonDrop parametrized complete comparison.**

A.6 Equations

A.6.1 Tree

Considering a tree with depth **D** and fan-out **F**.

$$\mathrm{N} = \frac{F^{D+1} - 1}{F - 1}, \quad \text{for } F > 1 \tag{A.1}$$

$$\mathrm{N} = D + 1, \quad \text{for } F = 1 \tag{A.2}$$

$$\mathrm{L} = F^{D} \tag{A.3}$$

$$\mathrm{I} = \frac{F^{D+1} - 1}{F - 1} - F^{D} \tag{A.4}$$

Figure A.35: (N) Number of nodes, (I) Number of Internal Nodes, (L) Number of Leaves.

Abbreviations

CDF	Cumulative Distribution Function. 41
CES	Central Exchange Server. 13
FAQ	Fast Approximate Quadratic Programming. 12
IaC	Infrastructure as Code. 23
IaaS	Infrastructure as a Service. 7
LOB	Limit Order Book. 13
ME	Matching Engine. 13
MPs	Market Participants. 13
OWD	one-way delay. 10
QAP	Quadratic Assignment Problem. 11
TSP	Time Series Plot. 41
VMs	Virtual Machines. 7, 9

Bibliography

[CEM07] Tatsuhiro Chiba, Toshio Endo, and Satoshi Matsuoka. High-performance mpi broadcast algorithm for grid environments utilizing multi-lane nics. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 487–494, 2007.

[Clo08] Google Cloud. Google cloud platform. `https://cloud.google.com/`, 2008. Accessed: 2024-09-12.

[Clo23] Clockwork. Clock sync. `https://www.clockwork.io/clock-sync/`, 2023. Accessed: 21-September-2024.

[GGM+23] Prateesh Goyal, Eashan Gupta, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. Dbo: Response time fairness for cloud-hosted financial exchanges, 2023.

[GGS+21] 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.

[GLR+24] Junzhi Gong, Yuliang Li, Devdeep Ray, KK Yap, and Nandita Dukkipati. Octopus: A fair packet delivery service, 2024.

[GLY+18] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosunblum, 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, page 81–94, USA, 2018. USENIX Association.

[Goo23] Google. Protocol buffers: Google's data interchange format, 2023. Accessed: 2023-09-14.

[GSPR22] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and high-performance consensus using synchronized clocks. *Proceedings of the VLDB Endowment*, 16(4):629–642, December 2022.

[Has23a] HashiCorp. Packer. https://www.packer.io/, 2023. Available at https://www.packer.io/.

[Has23b] HashiCorp. Terraform. https://www.terraform.io, 2023. Available at https://www.terraform.io.

[Hc23] Pieter Hintjens and contributors. Zeromq: Messaging for many applications, 2023. Available at https://zeromq.org/.

[HCW+24] Owen Hilyard, Bocheng Cui, Marielle Webster, Abishek Bangalore Muralikrishna, and Aleksey Charapko. Cloudy forecast: How predictable is communication latency in the cloud? In *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2024.

[HGB+24] Muhammad Haseeb, Jinkun Geng, Ulysses Butler, Xiyu Hao, Daniel Duclos-Cavalcanti, and Anirudh Sivaraman. Jasper: Scalable and fair multicast for financial exchanges in the cloud, 2024.

[IBM11] IBM. Ibm cloud. https://www.ibm.com/cloud/, 2011. Accessed: 2024-09-12.

[K+24] Holger Krekel et al. pytest: Simple powerful testing with python. https://docs.pytest.org/, 2004–2024. Version 7.x.

[KB57] Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957.

[lep24] leptonsys.com. Layer 1 switches: Key functions and technologies. https://www.leptonsys.com/blog/layer-1-switches-key-functions-andtechnologies, 2024.

[Loh24] Niels Lohmann. Json for modern c++. https://github.com/nlohmann/json, 2013–2024. Version 3.x.x.

[Mic10] Microsoft. Microsoft azure: Cloud computing services. https://azure.microsoft.com/, 2010. Accessed: 2024-09-12.

[Mic21] Microsoft. Microsoft azure: Cloud computing services. https://www.idc.com/getdoc.jsp?containerId=prUS48208321, 2021. Accessed: 2022-12-05.

[nov24] novasparks.com. Ultra low latency fpga market data solution. https://novasparks.com, 2024. Accessed: 2024-09-15.

[Sac22] Vighnesh Sachidananda. *Scheduling and Autoscaling Methods for Low Latency Applications*. Stanford University, 2022.

[Ser06] Amazon Web Services. Amazon ec2: Elastic compute cloud. https://aws.amazon.com/ec2/, 2006. Accessed: 2024-09-12.

[SST09] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.*, 35(12):581–594, dec 2009.

[VCL+14] Joshua T. Vogelstein, John M. Conroy, Vince Lyzinski, Louis J. Podrazik, Steven G. Kratzer, Eric T. Harley, Donniell E. Fishkind, R. Jacob Vogelstein, and Carey E. Priebe. Fast approximate quadratic programming for large (brain) graph matching, 2014.

[XMNB13] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, Lombard, IL, April 2013. USENIX Association.

[YAK14] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[YHG+17] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery.

[YHGK15] Neeraja Jayant Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, and Randy H. Katz. Faster jobs in distributed data processing using multi-task learning. In *SDM*, 2015.