JurreBrandsen1709/Hestia

Git Repository

Design of Hestia: a General Dyconit Middleware for Publish/Subscribe Systems

Jurre Brandsen

M.Sc. @ Vrije Universiteit, AtLarge Research

jurrebrandsen@gmail.com

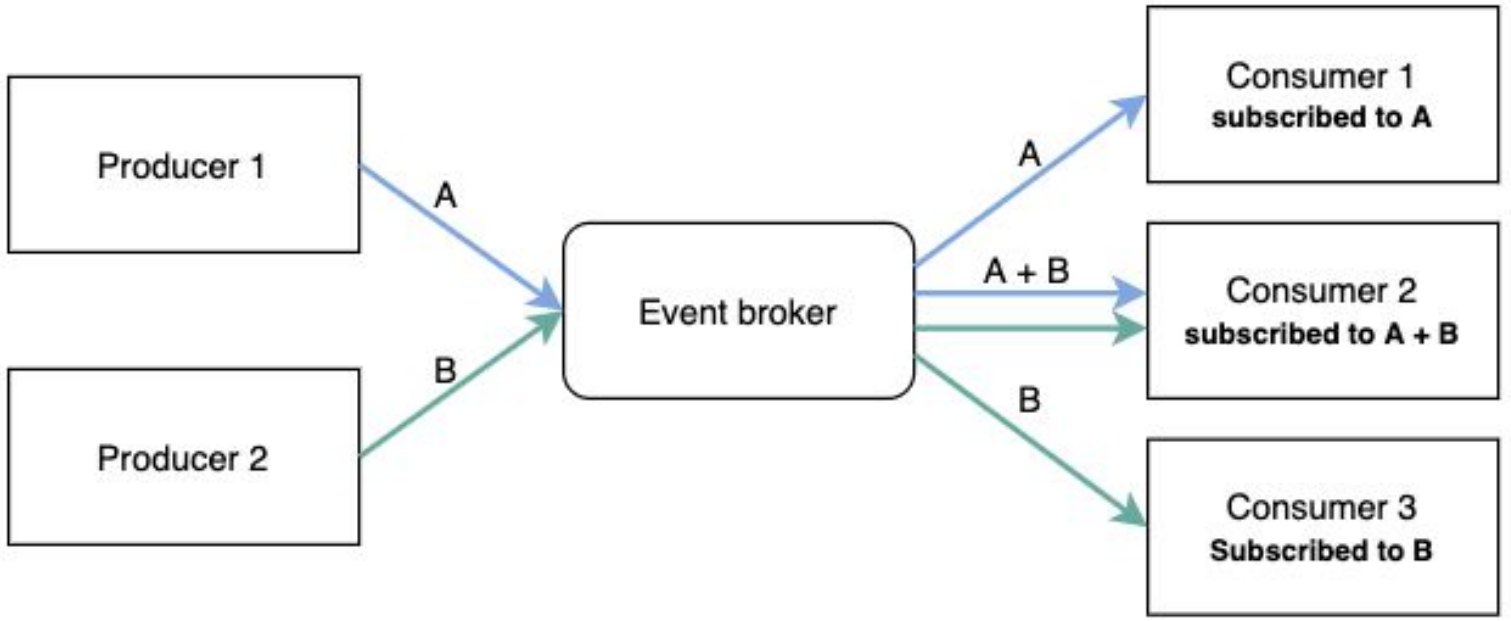
https://atlarge-research.com/
https://research.infosupport.com/

Ir. Jesse Donkervliet

Rinse van Hees (Info Support)

Introduction: event-driven systems & publish/subscribe

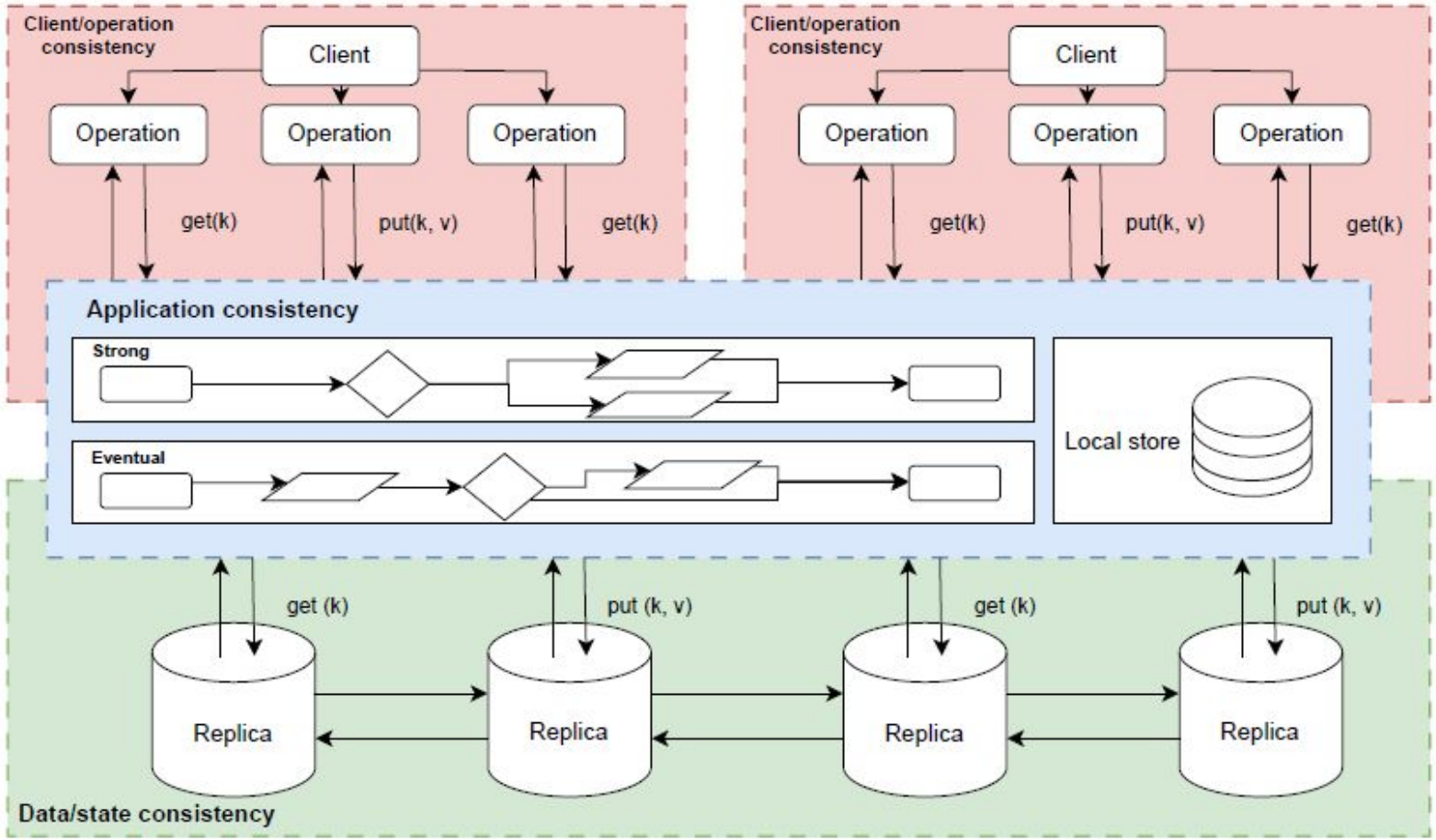
- A system that reacts to events
  - messages, sensor reading, etc
- Loosely coupled
- scalable, resilient and agile



Introduction: Application-Centric Consistency

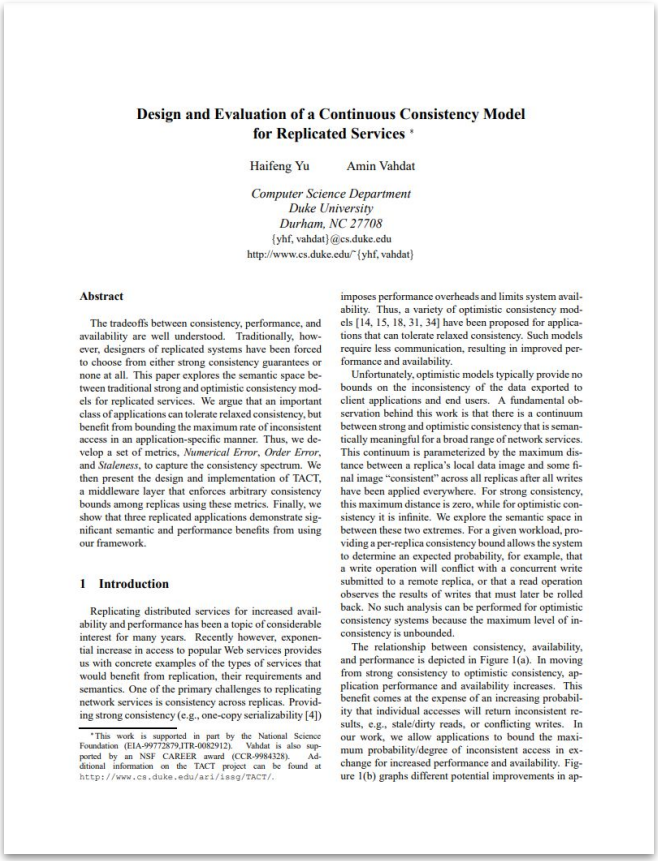
Application-centric:

- tailored to the needs of the application
- Fine-grained consistency



Introduction: Continuous Consistency

- A **consistency model** that captures the **spectrum** between **strong** and **optimistic** consistency for replicated services.
- Allows applications to **specify** and **enforce** their own **consistency semantics** using three metrics: **numerical error**, **order error**, and **staleness**.
- A **conit** is a unit of consistency that represents a set of data items that share the same consistency requirements.



Design and Evaluation of a Continuous Consistency Model for Replicated Services *

Haifeng Yu Amin Vahdat

Computer Science Department
Duke University
Durham, NC 27708
{yhf, vahdat}@cs.duke.edu
http://www.cs.duke.edu/~{yhf, vahdat}

Abstract

The tradeoffs between consistency, performance, and availability are well understood. Traditionally, however, designers of replicated systems have been forced to choose from either strong consistency guarantees or none at all. This paper explores the semantic space between traditional strong and optimistic consistency models for replicated services. We argue that an important class of applications can tolerate relaxed consistency, but benefit from bounding the maximum rate of inconsistent access in an application-specific manner. Thus, we develop a set of metrics, *Numerical Error*, *Order Error*, and *Staleness*, to capture the consistency spectrum. We then present the design and implementation of TACT, a middleware layer that enforces arbitrary consistency bounds among replicas using these metrics. Finally, we show that three replicated applications demonstrate significant semantic and performance benefits from using our framework.

1 Introduction

Replicating distributed services for increased availability and performance has been a topic of considerable interest for many years. Recently however, exponential increase in access to popular Web services provides us with concrete examples of the types of services that would benefit from replication, their requirements and semantics. One of the primary challenges to replicating network services is consistency across replicas. Providing strong consistency (e.g., one-copy serializability [4]) imposes performance overheads and limits system availability. Thus, a variety of optimistic consistency models [14, 15, 18, 31, 34] have been proposed for applications that can tolerate relaxed consistency. Such models require less communication, resulting in improved performance and availability.

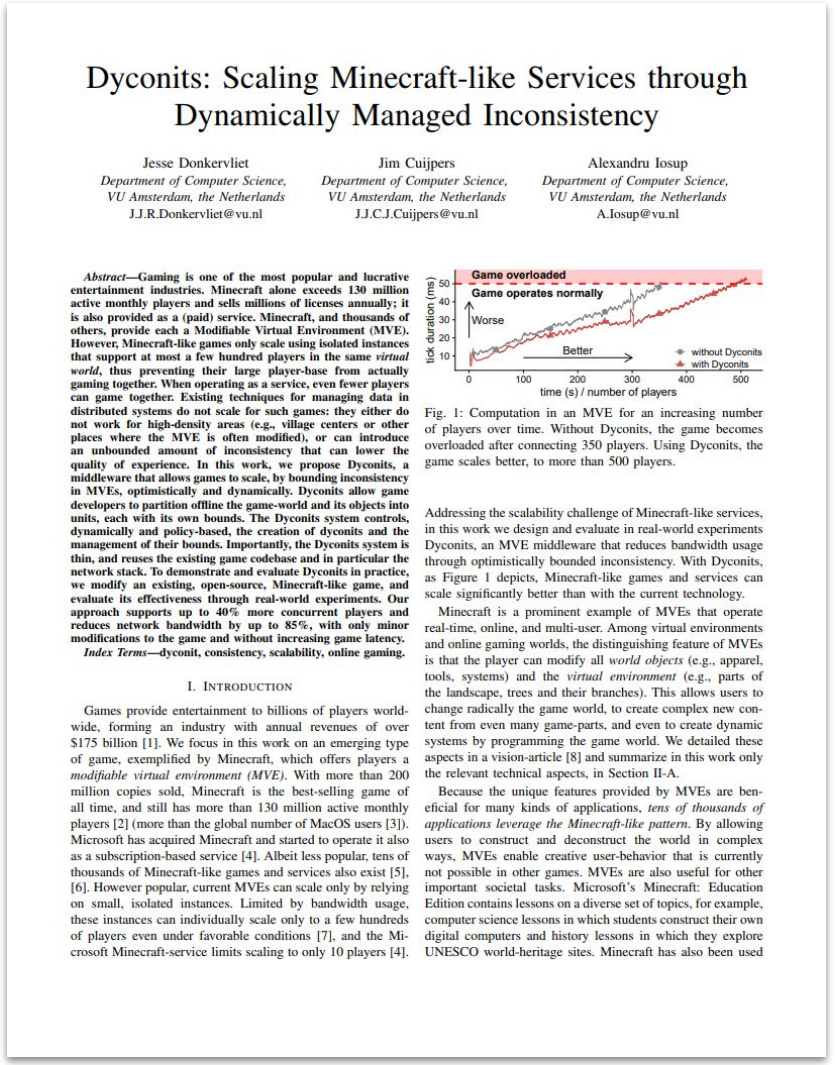
Unfortunately, optimistic models typically provide no bounds on the inconsistency of the data exported to client applications and end users. A fundamental observation behind this work is that there is a continuum between strong and optimistic consistency that is semantically meaningful for a broad range of network services. This continuum is parameterized by the maximum distance between a replica's local data image and some final image "consistent" across all replicas after all writes have been applied everywhere. For strong consistency, this maximum distance is zero, while for optimistic consistency it is infinite. We explore the semantic space in between these two extremes. For a given workload, providing a per-replica consistency bound allows the system to determine an expected probability, for example, that a write operation will conflict with a concurrent write submitted to a remote replica, or that a read operation observes the results of writes that must later be rolled back. No such analysis can be performed for optimistic consistency systems because the maximum level of inconsistency is unbounded.

The relationship between consistency, availability, and performance is depicted in Figure 1(a). In moving from strong consistency to optimistic consistency, application performance and availability increases. This benefit comes at the expense of an increasing probability that individual accesses will return inconsistent results, e.g., stale/dirty reads, or conflicting writes. In our work, we allow applications to bound the maximum probability/degree of inconsistent access in exchange for increased performance and availability. Figure 1(b) graphs different potential improvements in ap-

*This work is supported in part by the National Science Foundation (EIA-9972879,ITR-0082912). Vahdat is also supported by an NSF CAREER award (CCR-9984328). Additional information on the TACT project can be found at http://www.cs.duke.edu/ari/issg/TACT/.

Introduction: Dynamic Consistency Units (Dyconits)

- A consistency model that captures the **spectrum** between **strong** and **optimistic** consistency for **replicated services**.
- Allows applications to **specify** and **enforce** their own consistency semantics using three metrics: **numerical error, order error, and staleness**.
- Introduces Dynamic Consistency Units
- Modifiable Virtual Environments (MVE)
- Results positive:
  - 40% increase in concurrent players
  - 85% reduction in bandwidth usage



Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency

Jesse Donkervliet
Department of Computer Science, VU Amsterdam, the Netherlands
J.J.R.Donkervliet@vu.nl

Jim Cuijpers
Department of Computer Science, VU Amsterdam, the Netherlands
J.J.C.J.Cuijpers@vu.nl

Alexandru Iosup
Department of Computer Science, VU Amsterdam, the Netherlands
A.Iosup@vu.nl

***Abstract*—Gaming is one of the most popular and lucrative entertainment industries. Minecraft alone exceeds 130 million active monthly players and sells millions of licenses annually; it is also provided as a (paid) service. Minecraft, and thousands of others, provide each a Modifiable Virtual Environment (MVE). However, Minecraft-like games only scale using isolated instances that support at most a few hundred players in the same *virtual world*, thus preventing their large player-base from actually gaming together. When operating as a service, even fewer players can game together. Existing techniques for managing data in distributed systems do not scale for such games: they either do not work for high-density areas (e.g., village centers or other places where the MVE is often modified), or can introduce an unbounded amount of inconsistency that can lower the quality of experience. In this work, we propose Dyconits, a middleware that allows games to scale, by bounding inconsistency in MVEs, optimistically and dynamically. Dyconits allow game developers to partition offline the game-world and its objects into units, each with its own bounds. The Dyconits system controls, dynamically and policy-based, the creation of dyconits and the management of their bounds. Importantly, the Dyconits system is thin, and reuses the existing game codebase and in particular the network stack. To demonstrate and evaluate Dyconits in practice, we modify an existing, open-source, Minecraft-like game, and evaluate its effectiveness through real-world experiments. Our approach supports up to 40% more concurrent players and reduces network bandwidth by up to 85%, with only minor modifications to the game and without increasing game latency.**

***Index Terms*—dyconit, consistency, scalability, online gaming.**

Fig. 1: Computation in an MVE for an increasing number of players over time. Without Dyconits, the game becomes overloaded after connecting 350 players. Using Dyconits, the game scales better, to more than 500 players.

I. Introduction

Games provide entertainment to billions of players worldwide, forming an industry with annual revenues of over $175 billion [1]. We focus in this work on an emerging type of game, exemplified by Minecraft, which offers players a *modifiable virtual environment (MVE)*. With more than 200 million copies sold, Minecraft is the best-selling game of all time, and still has more than 130 million active monthly players [2] (more than the global number of MacOS users [3]). Microsoft has acquired Minecraft and started to operate it also as a subscription-based service [4]. Albeit less popular, tens of thousands of Minecraft-like games and services also exist [5], [6]. However popular, current MVEs can scale only by relying on small, isolated instances. Limited by bandwidth usage, these instances can individually scale only to a few hundreds of players even under favorable conditions [7], and the Microsoft Minecraft-service limits scaling to only 10 players [4].

Addressing the scalability challenge of Minecraft-like services, in this work we design and evaluate in real-world experiments Dyconits, an MVE middleware that reduces bandwidth usage through optimistically bounded inconsistency. With Dyconits, as Figure 1 depicts, Minecraft-like games and services can scale significantly better than with the current technology.

Minecraft is a prominent example of MVEs that operate real-time, online, and multi-user. Among virtual environments and online gaming worlds, the distinguishing feature of MVEs is that the player can modify all *world objects* (e.g., apparel, tools, systems) and the *virtual environment* (e.g., parts of the landscape, trees and their branches). This allows users to change radically the game world, to create complex new content from even many game-parts, and even to create dynamic systems by programming the game world. We detailed these aspects in a vision-article [8] and summarize in this work only the relevant technical aspects, in Section II-A.

Because the unique features provided by MVEs are beneficial for many kinds of applications, *tens of thousands of applications leverage the Minecraft-like pattern*. By allowing users to construct and deconstruct the world in complex ways, MVEs enable creative user-behavior that is currently not possible in other games. MVEs are also useful for other important societal tasks. Microsoft's Minecraft: Education Edition contains lessons on a diverse set of topics, for example, computer science lessons in which students construct their own digital computers and history lessons in which they explore UNESCO world-heritage sites. Minecraft has also been used

Problem statement

- Application-centric consistency models on the rise
- Dyconits show promising results in MVEs
- Need for general library

Goal:

1. Assessing Dyconits' generalisability
2. Exploring optimal utilisation strategies

Scope:

Event-driven systems in which consumers subscribe to the same topic and can share processed work

Research Questions

RQ1 – What are common requirements in event-driven systems that can inform the design of a generic Dyconit system?

RQ2 – How to design a generic Dyconit system?

RQ3 – How to integrate a generic Dyconit system into event-driven systems?

RQ4 – How to evaluate a generic Dyconit system for event-driven systems?

Requirement Analysis

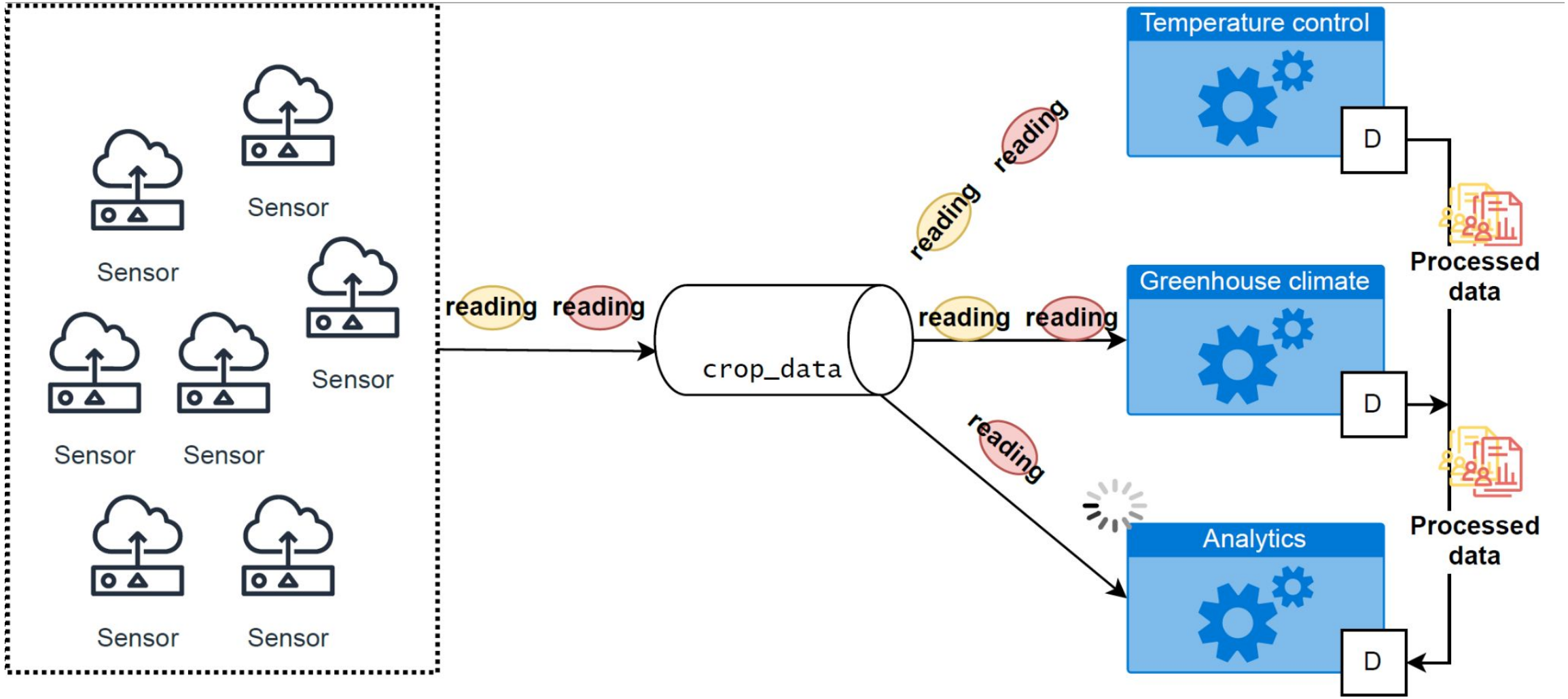
Methodology:

Informal discussions with domain experts at Info Support

Domains:

Smart IoT farming, Online Messaging and Real-time analytics

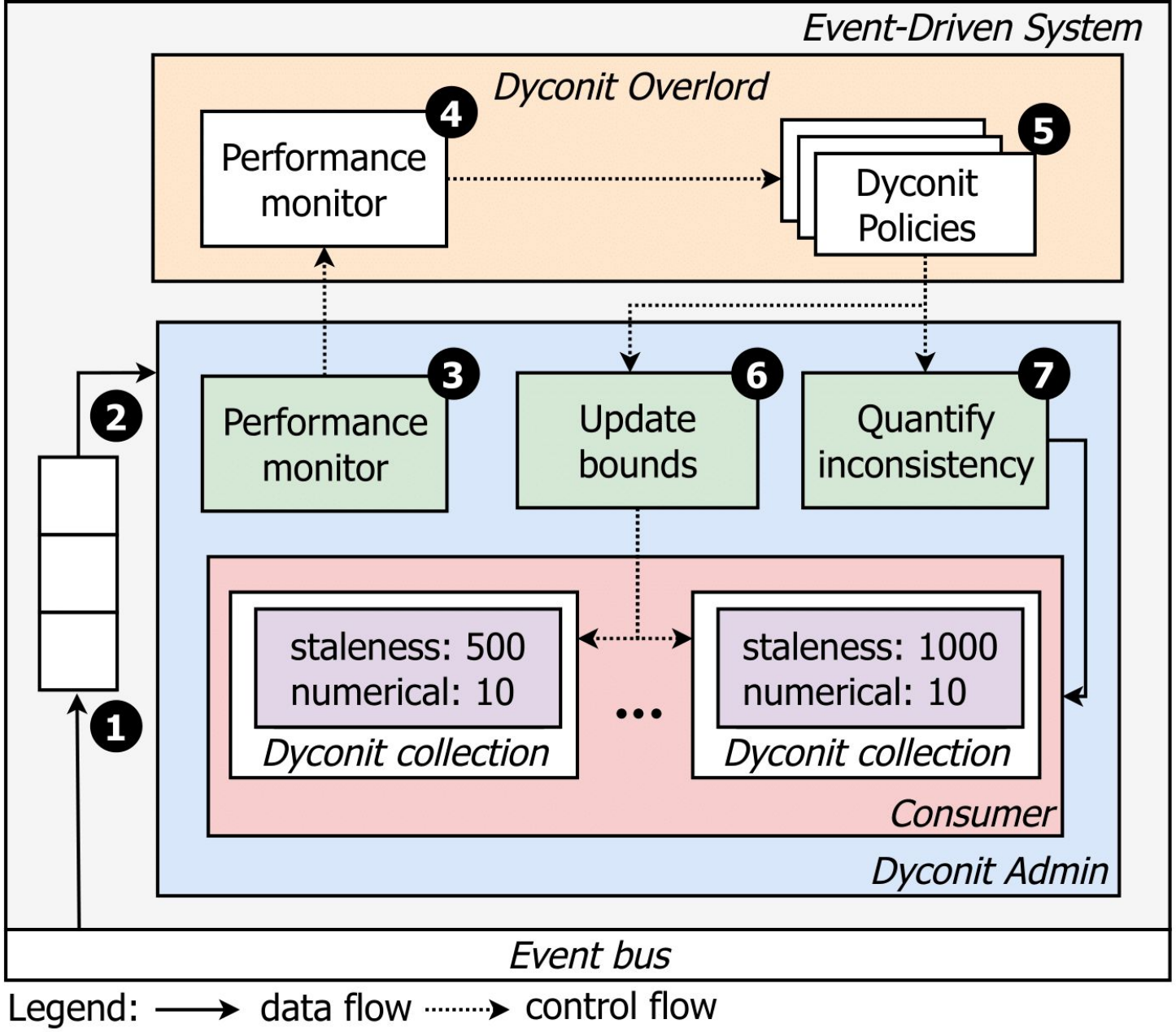
Example application: smart IoT farming



Design

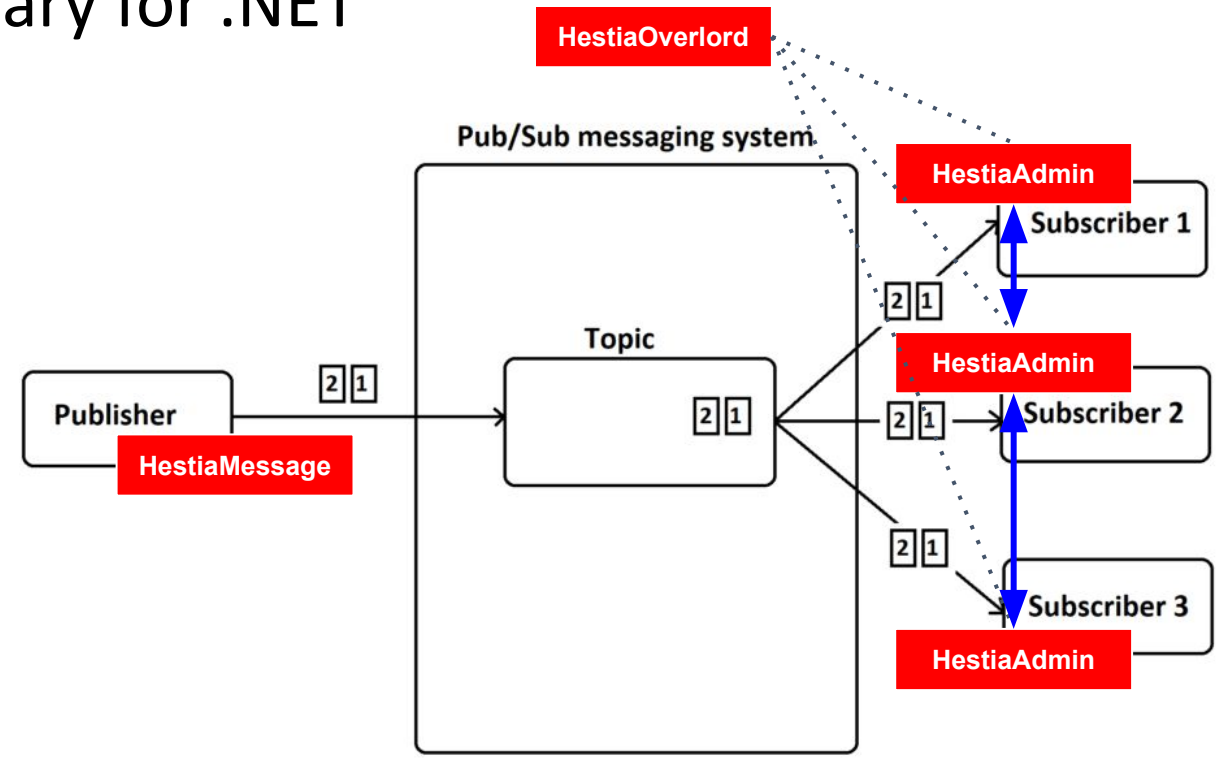
Dyconit Overlord:
Assumes responsibility of maintaining an overview of the system

Dyconit Admin:
Middleware that enables dyconits in EDAs



Implementation: Introducing Hestia

Confluent.Kafka library for .NET

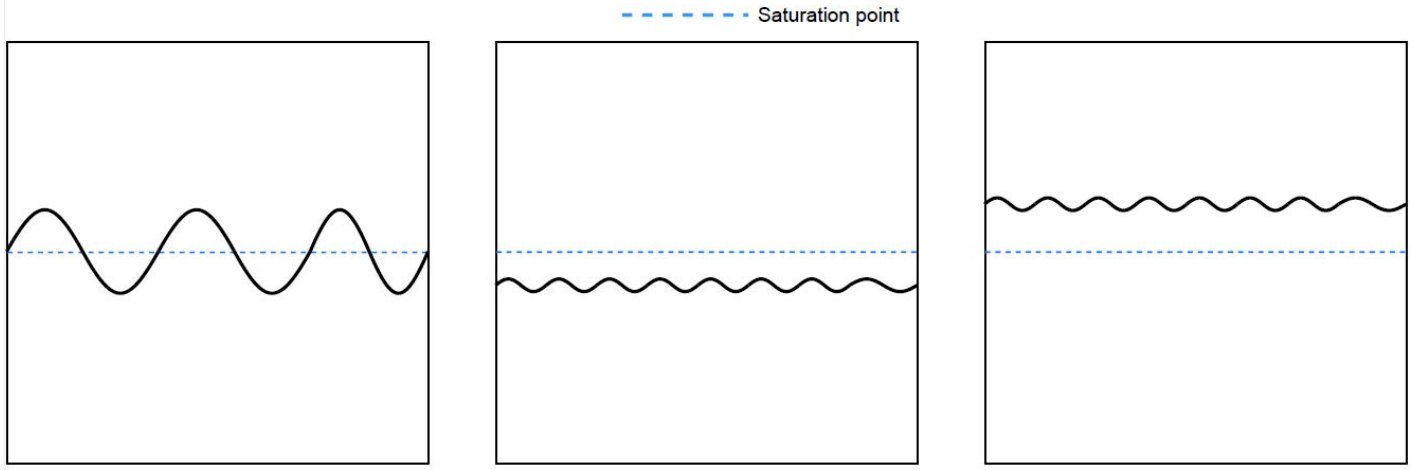


Evaluation Setup: Designing Workloads

Interviewed domain experts from various fields at Info Support to gain insights into real-world workload patterns

Workloads:

- Periods of high and steady event rates, sudden peaks and downtime
- 80/20 normal event/priority event ratio
- Priority event is larger than normal events by a factor of 20



	Topic Priority	Topic Normal
Staleness	10 s	20 s
Numerical Error	10	25
Message weight	1	1

Evaluation Setup: Metrics

Table 6.1: The definitions and units of the five metrics used to evaluate the performance of our system in our experiments: consumer lag (CL), message throughput (MT), overhead throughput (OT)

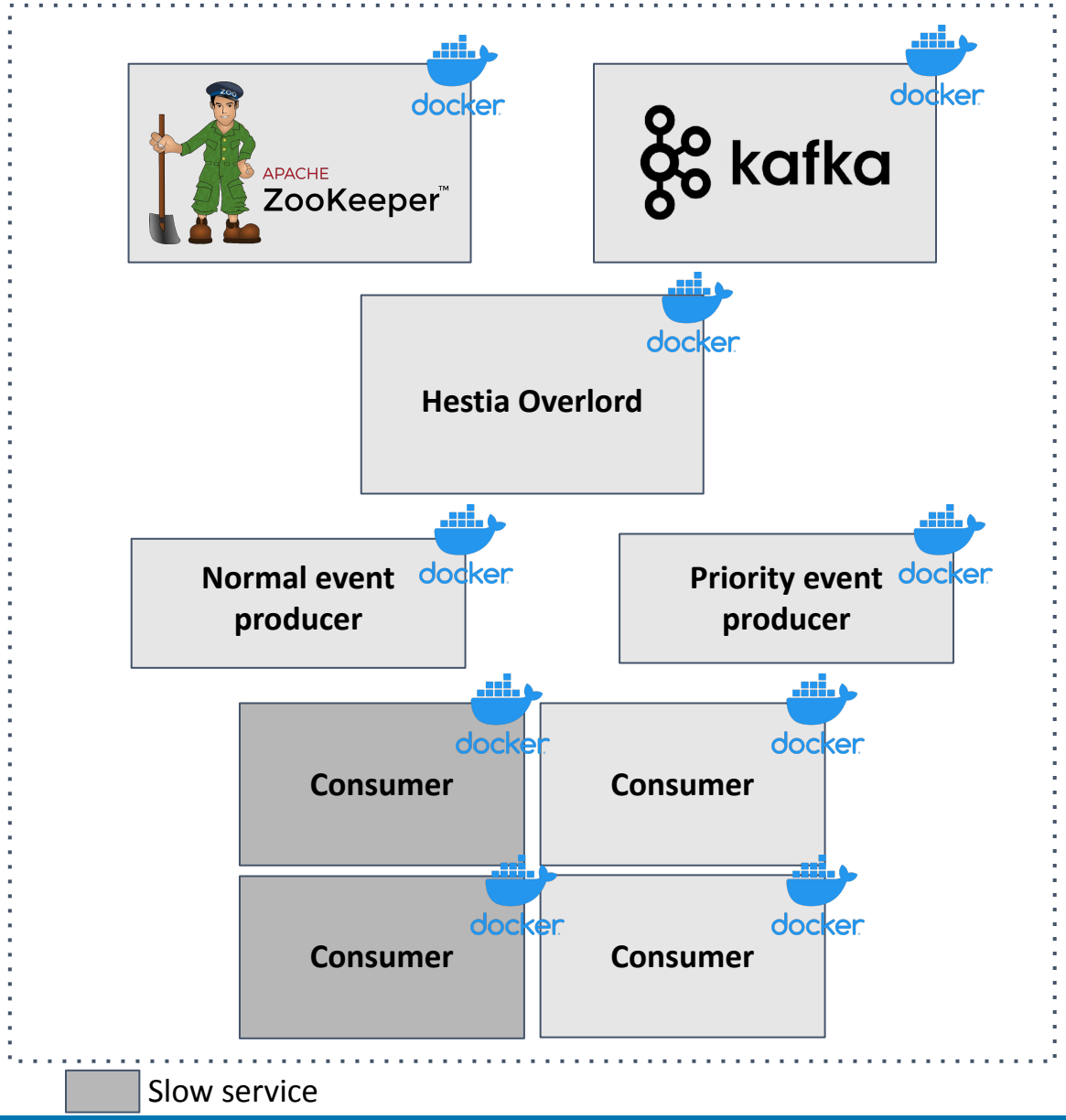
Metric	Definition	Unit
Consumer lag (CL)	The maximum delay between any two services	Seconds
Message throughput (MT)	The number of event-related messages sent or received by the system per second	Messages/second
Overhead throughput (OT)	The number of synchronisation-related messages sent or received by the system	Messages/second

Evaluation Deployment: Simulating real-world

- Added predefined uniform processing delay for consumers to simulate latency
  - Between 300 - 900 ms for slow services (2/4)
  - Between 200 - 600 ms for normal operating services (2/4)

CPU	12th Gen Intel Core i7-12800H
RAM	32 GiB

Hardware setup



Experiment Design

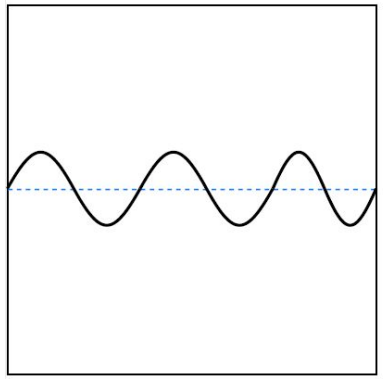
Table 6.2: Experiment configurations.
(Notation: MT = Message Throughput, OT = Overhead Throughput, CL = Consumer Lag)

Experiment	Topology	Workload	Policies	Metrics
Group 1: Performance Evaluation				
§6.6.1 Dyconit Impact	T1	W2	P1	CL, MT
Group 2: System Configuration Evaluation				
§6.6.2 Topology Impact	T1,T2	W2	P1	CL, MT, OT
§6.6.3 Workload Impact	T1	W1, W2, W3	P1	CL, MT, OT
§6.6.4 Policy Impact	T1	W2	none, P1, P2, P3	CL, MT, OT
Group 3: System Behaviour Evaluation				
§6.6.5 Sensitivity Analysis	T1	W2	dynamic range	CL, MT, OT

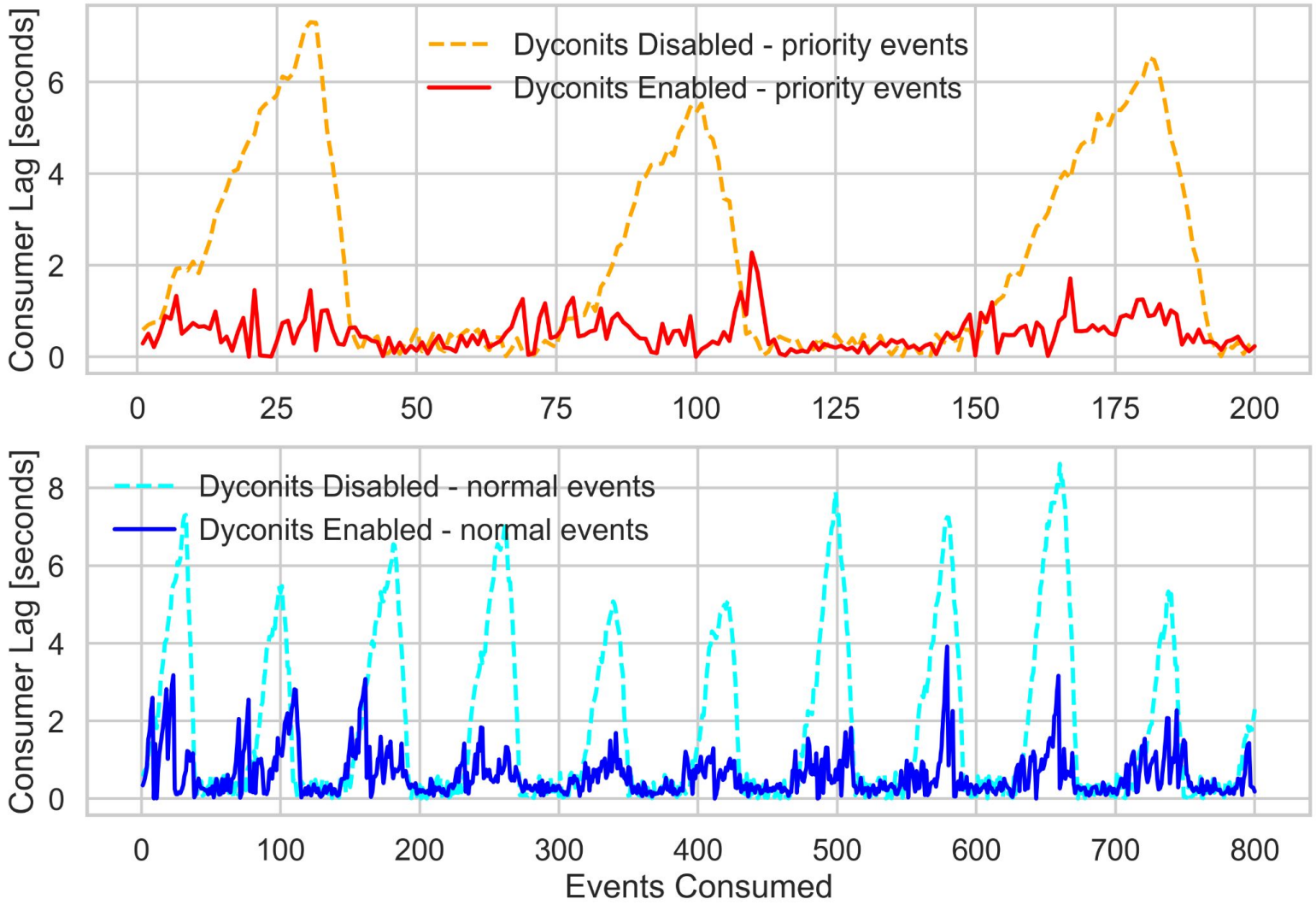
Experimental Evaluations: Optimistic Inconsistency

MF1 – By using Dyconits, Hestia reduces inconsistency by up to 70%, effectively bounding inconsistency

MF2 – Hestia can selectively influence the consistency level of different message types and prioritise certain message types over others.

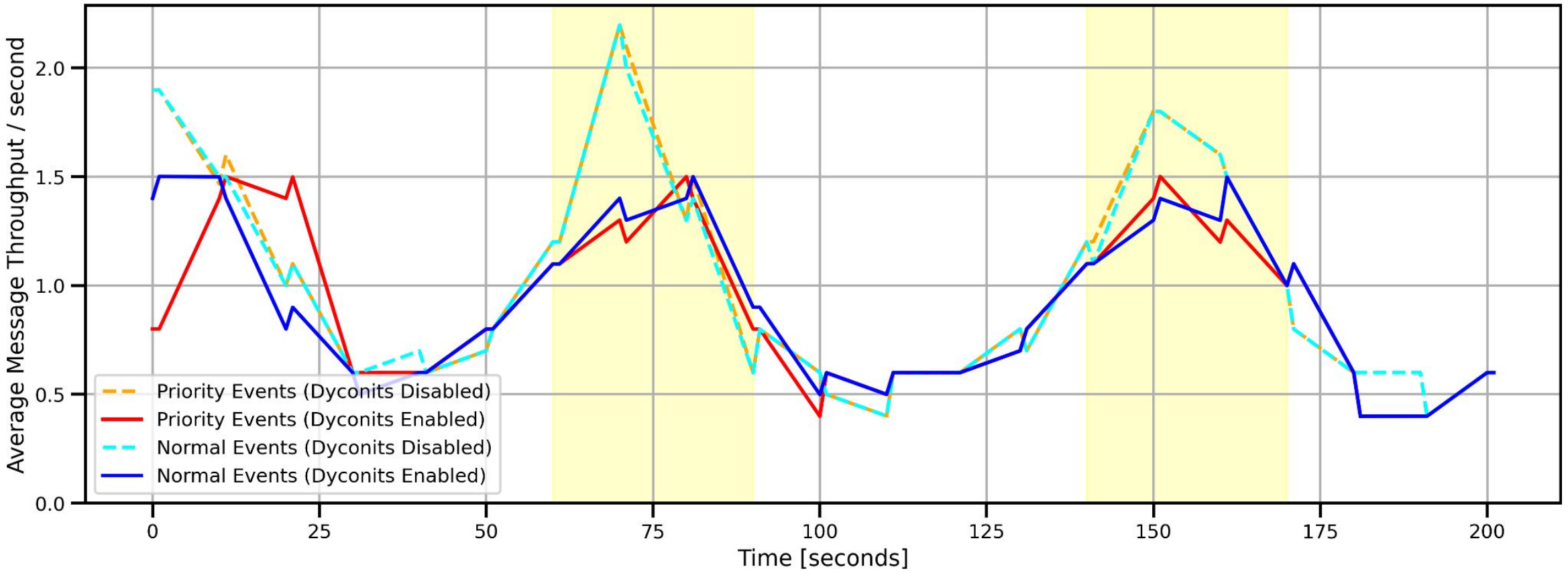


Fluctuating workload



Experimental Evaluations: Optimistic Inconsistency

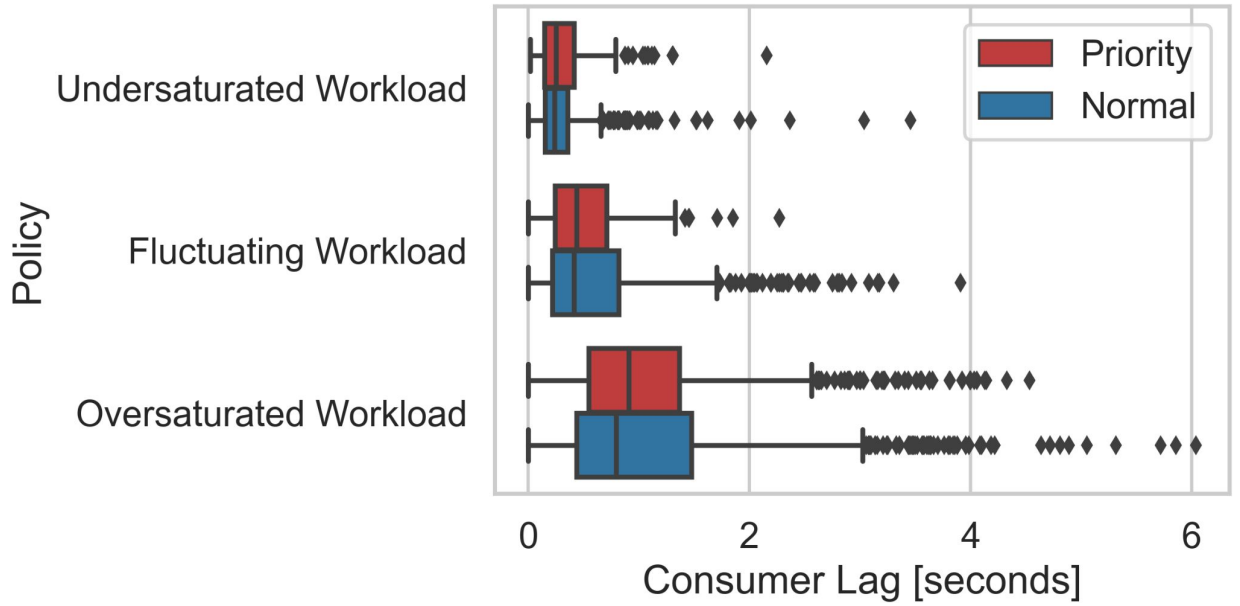
MF3 – Hestia introduces additional overhead, lowering the maximum throughput it can reach by approximately 45% compared to the baseline system without Dyconits.

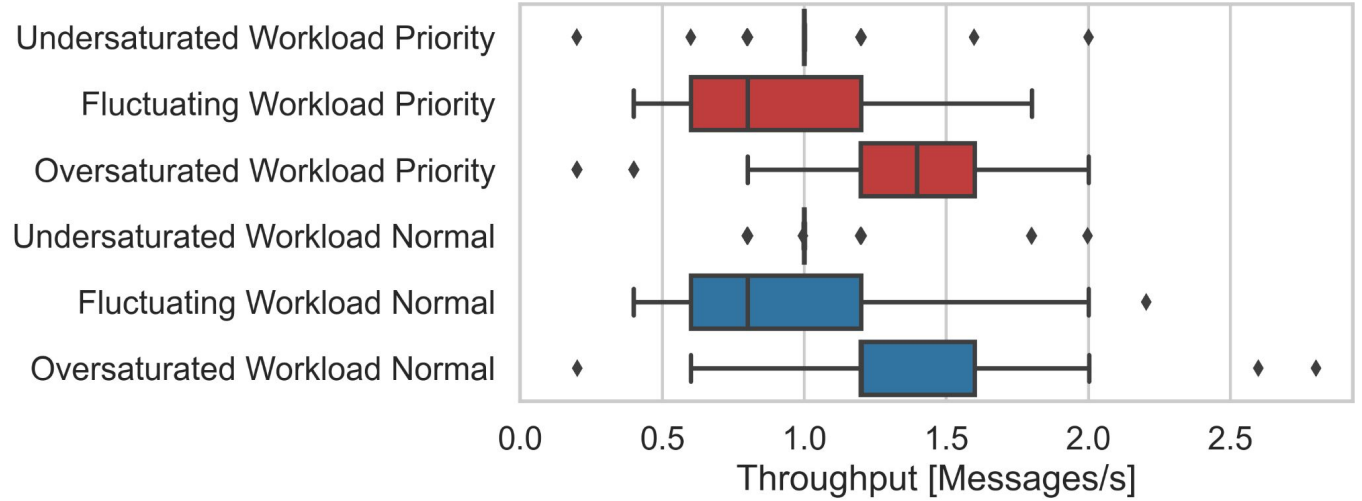


Experimental Evaluations: Varied Workloads

MF6 – Hestia is adaptable to different workloads, illustrating generality in design and implementation

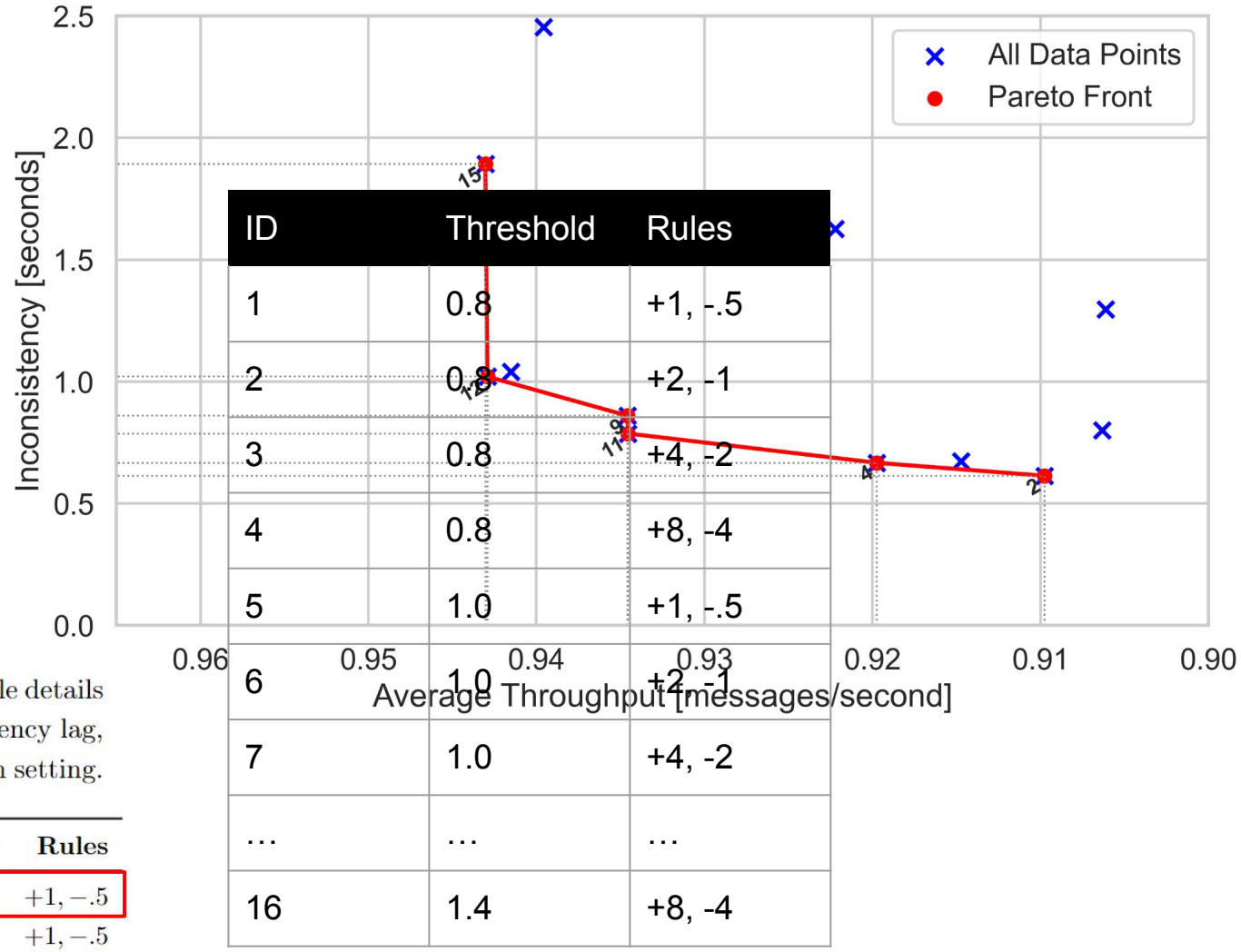
MF7 – Hestia adapts to workload saturation by balancing performance and consistency. We boost performance if the workload is oversaturated and enhance consistency at the cost of performance at undersaturation.





Experimental Evaluations: Sensitivity Analysis

MF10 – We observe that loose bounds offer a balance of weak consistency and high performance, while tight bounds ensure strong consistency but lower performance.



ID	Threshold	Rules
1	0.8	+1, -.5
2	0.8	+2, -1
3	0.8	+4, -2
4	0.8	+8, -4
5	1.0	+1, -.5
6	1.0	+2, -1
7	1.0	+4, -2
…	…	…
16	1.4	+8, -4

Table 6.8: Optimal settings for priority events under fluctuating workloads. The table details the most efficient configurations, showcasing their corresponding maximum inconsistency lag, event and overhead throughputs, as well as the thresholds and rules employed for each setting.

Policy ID	Lag (ms)	Message Throughput	Overhead throughput	Threshold	Rules
2	613	0.909759	14.796	1.0	+1, −.5
4	667	0.919751	14.312	1.4	+1, −.5
11	786	0.934547	8.467	1.2	+4, −2
9	860	0.934584	7.412	0.8	+4, −2
12	1020	0.942914	10.193	1.4	+4, −2
15	1893	0.943043	6.068	1.2	+8, −4

Take-Home Message

Analysis: Shared requirements in agriculture, finance, and messaging sectors identified.

Design: Dyconit extension created for event-driven systems, addressing replica divergence.

Implementation: “Hestia” introduced for Dyconit consistency in event-driven environments, offering optimistic consistency and adaptable policies.

Experiments: Hestia allows users to balance consistency, throughput, and overhead based on preferences and workload.

Future Work: Application-centric consistency models, real-world experiments with Hestia, and exploration of alternative implementations.

Presentation

Thesis

Code and Results:
JurreBrandsen1709/Hestia

Jurre Brandsen
M.Sc. @ Vrije Universiteit, AtLarge Research, InfoSupport
jurrebrandsen@gmail.com

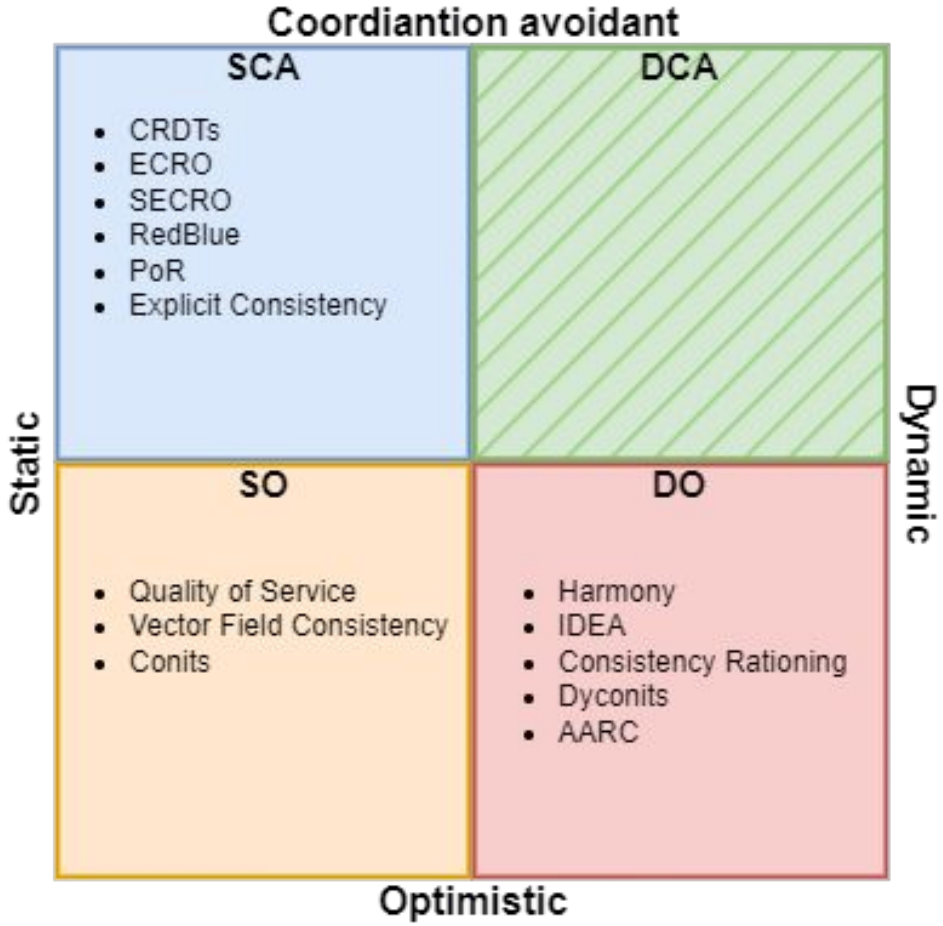
Backup slides

Design: Consistency Model Classification

Original Dyconits categorised under **dynamic/optimistic subcategory**

The ‘general Dyconit system’ does too:

- It can control consistency at the application level
- Specifies boundaries that permit to control consistency at the application level
- Incorporates a mechanism for dynamically adjusting bounds according to policies



Taxonomy of application-centric consistency models

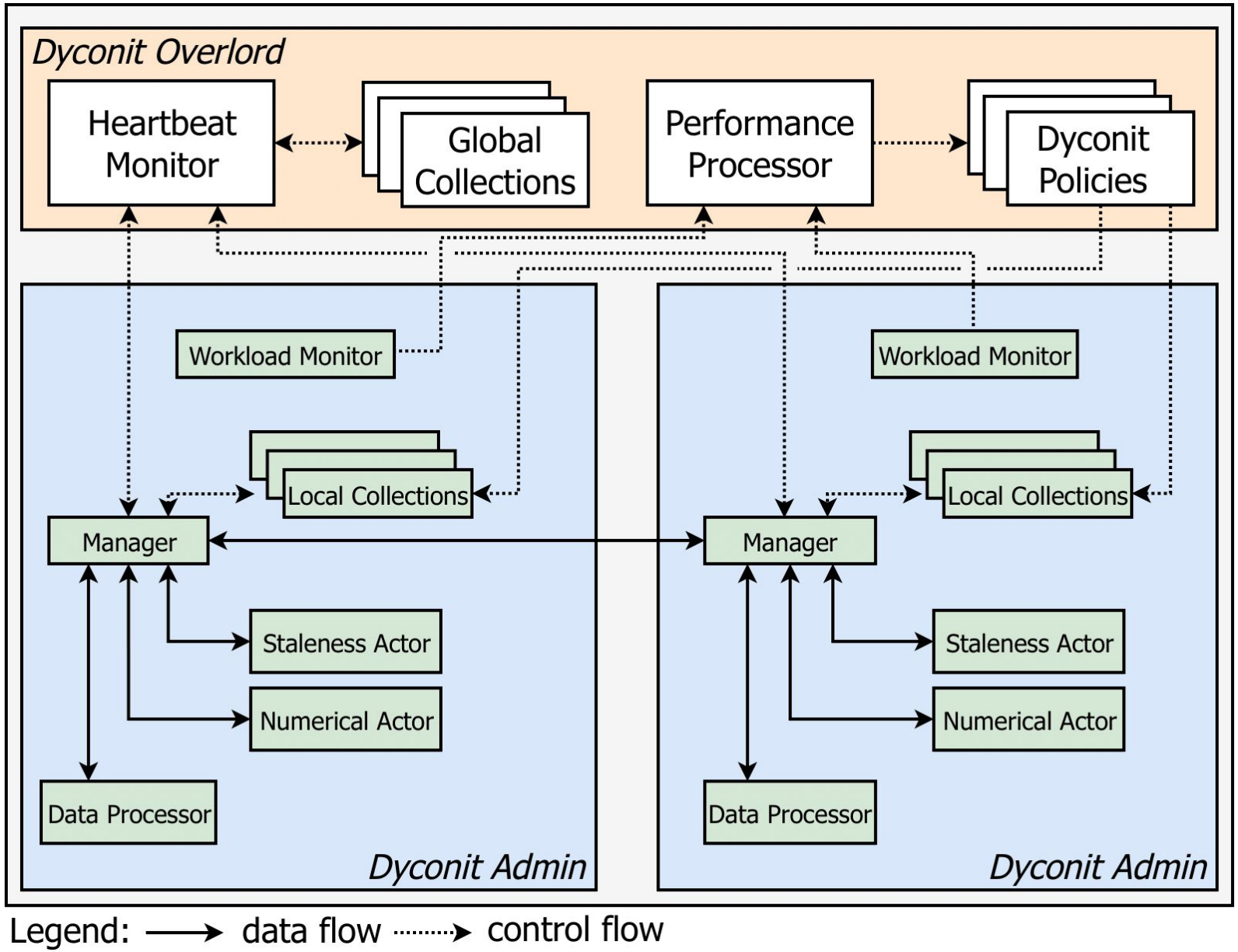
Design: detailed

The Dyconit Overlord

- monitoring the heartbeat of the nodes
- storing the global collections of dyconits
- processing the performance metrics
- applying the dyconit policies.

The Dyconit Admin

- managing the local collections of dyconits
- bounding the inconsistency in staleness and numerical dimensions
- processing the data from other nodes.



Design: Consistency Bounding

Staleness:

Each node keeps track of the last time writes were pulled from each node in the collection.

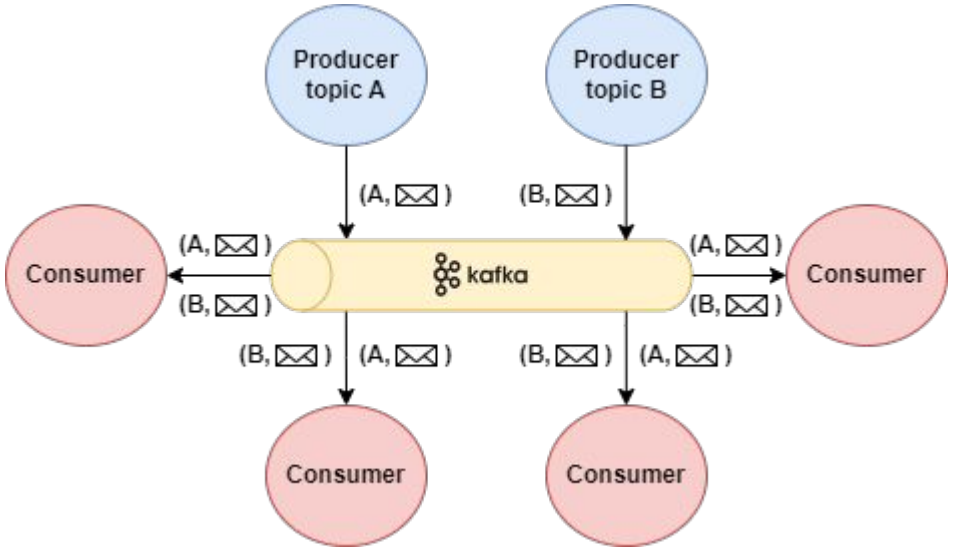
$$E_r = \max_{i=1}^{n}(t_i - T_i^r)$$

Numerical Error:

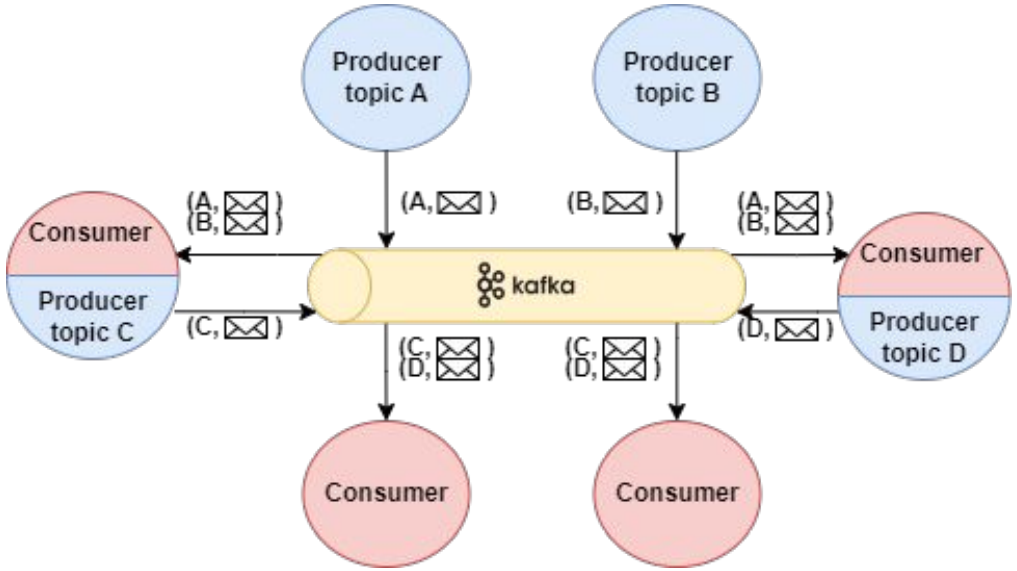
The numerical error of a node is defined by first assigning a global weight to each write, and then calculating the sum of the weights of locally unseen writes.

$$E_i^j = \sum_{k \in U_i^j} w_k$$

Evaluation Setup: Communication Patterns



Star topology

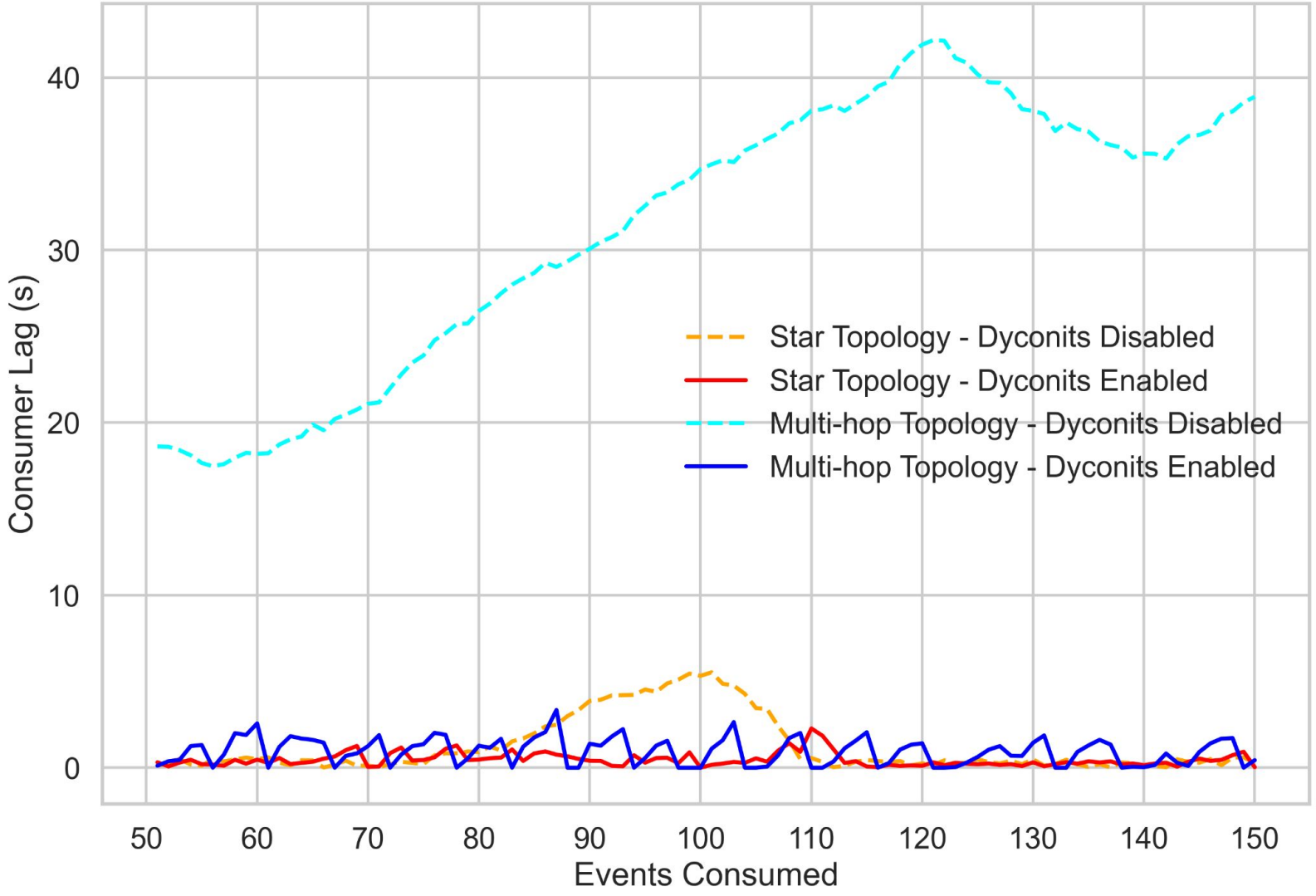


Multi-hop topology

Experimental Evaluations: Topology analysis

MF4 – Hestia is applicable to two common topologies that share the same basic building blocks, thus generalising to other communication patterns

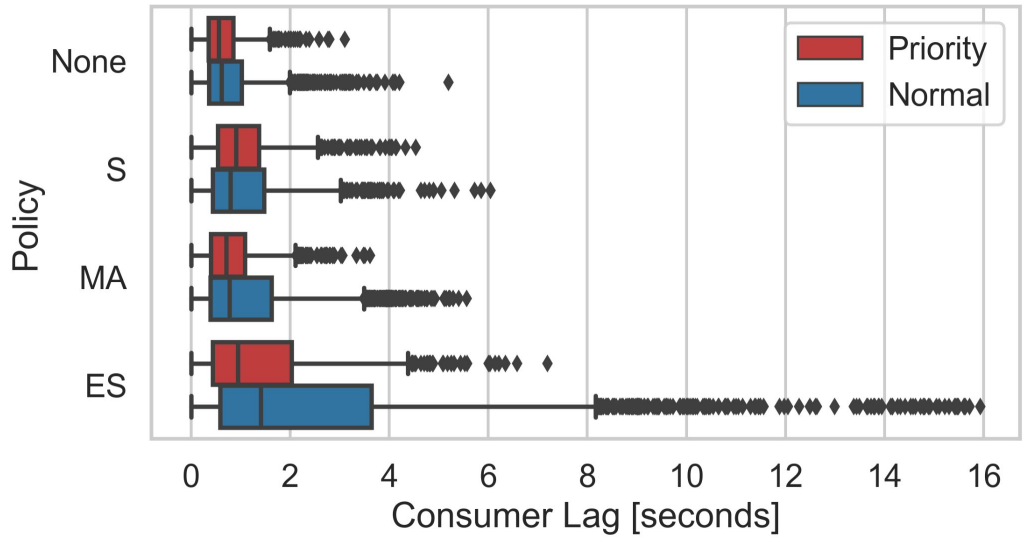
MF5 – Hestia achieves a significant reduction in consumer lag in multi-top topologies, lowering the lag by a factor of 40.

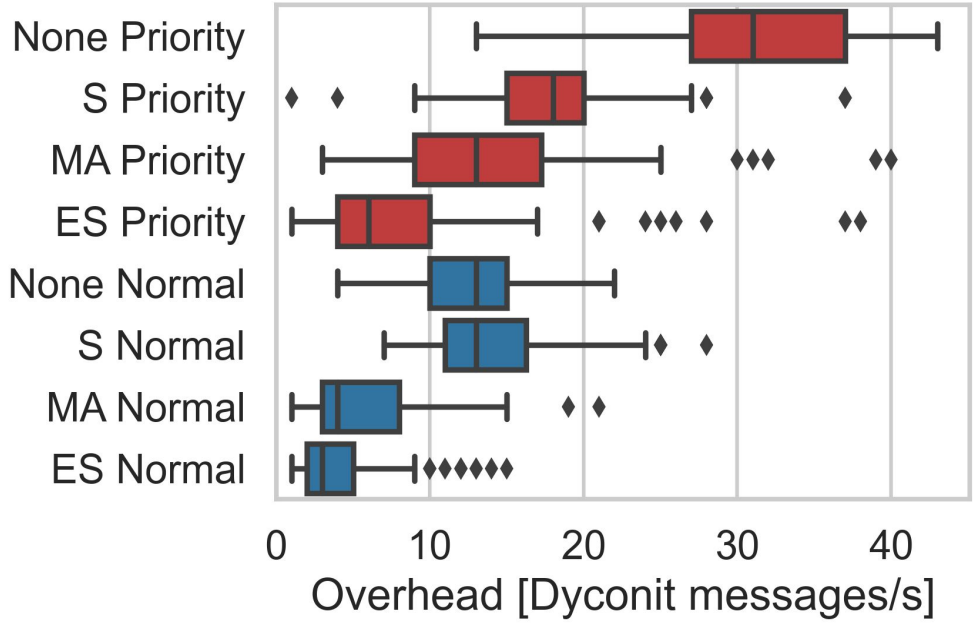


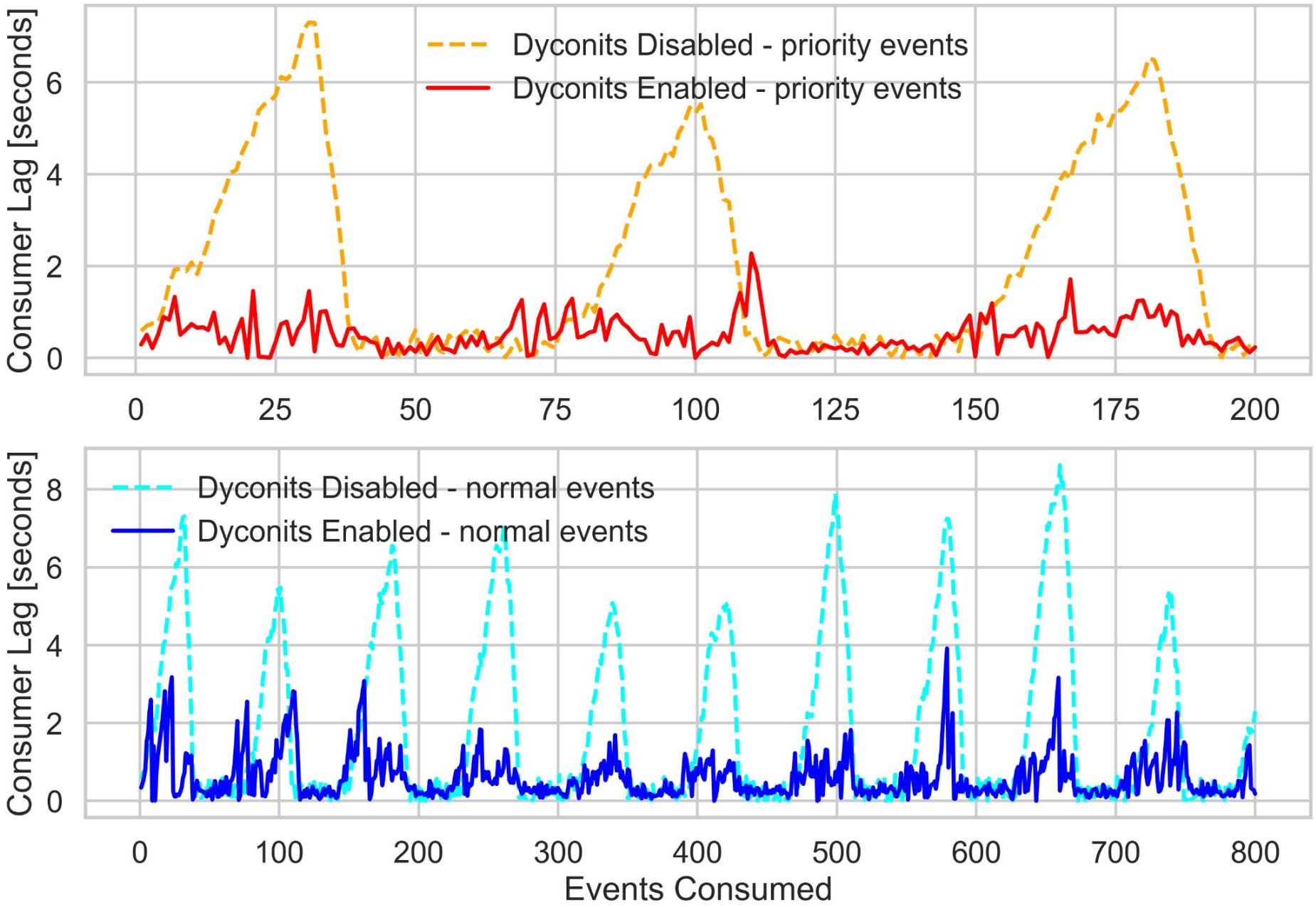
Experimental Evaluations: Varied Policies

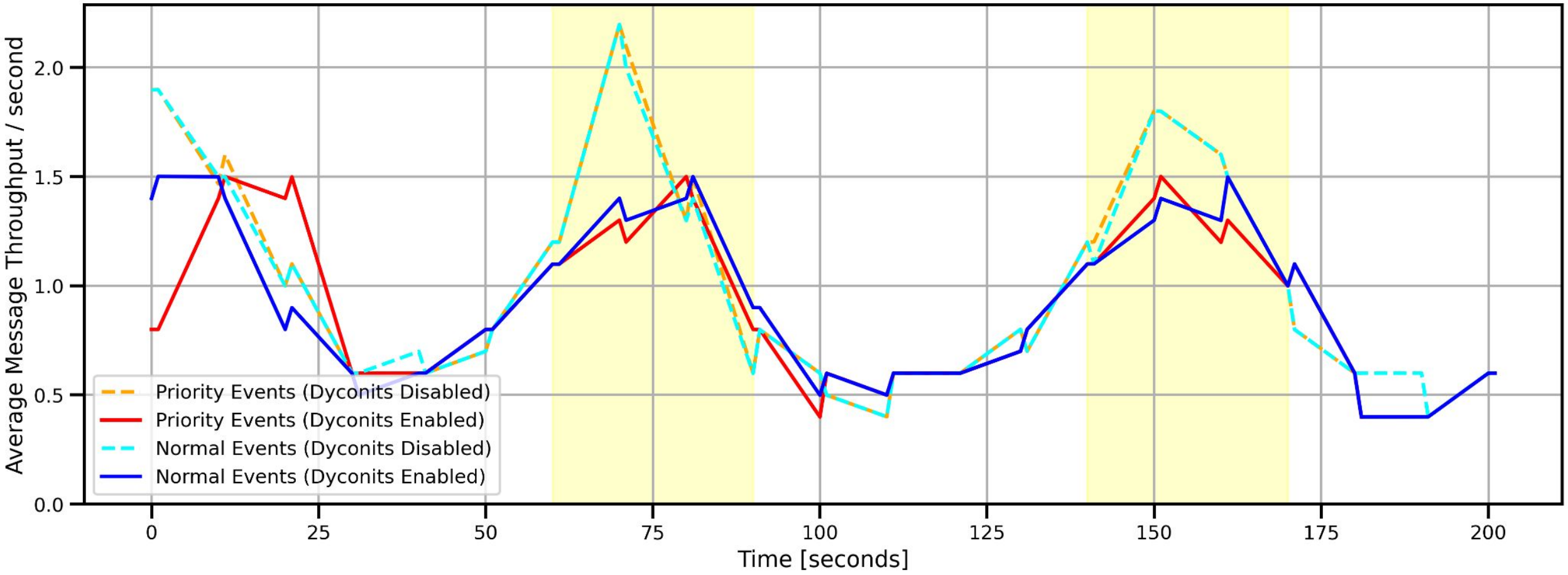
MF8 – Hestia is a flexible and customizable system that can adapt to different applications and user preferences.

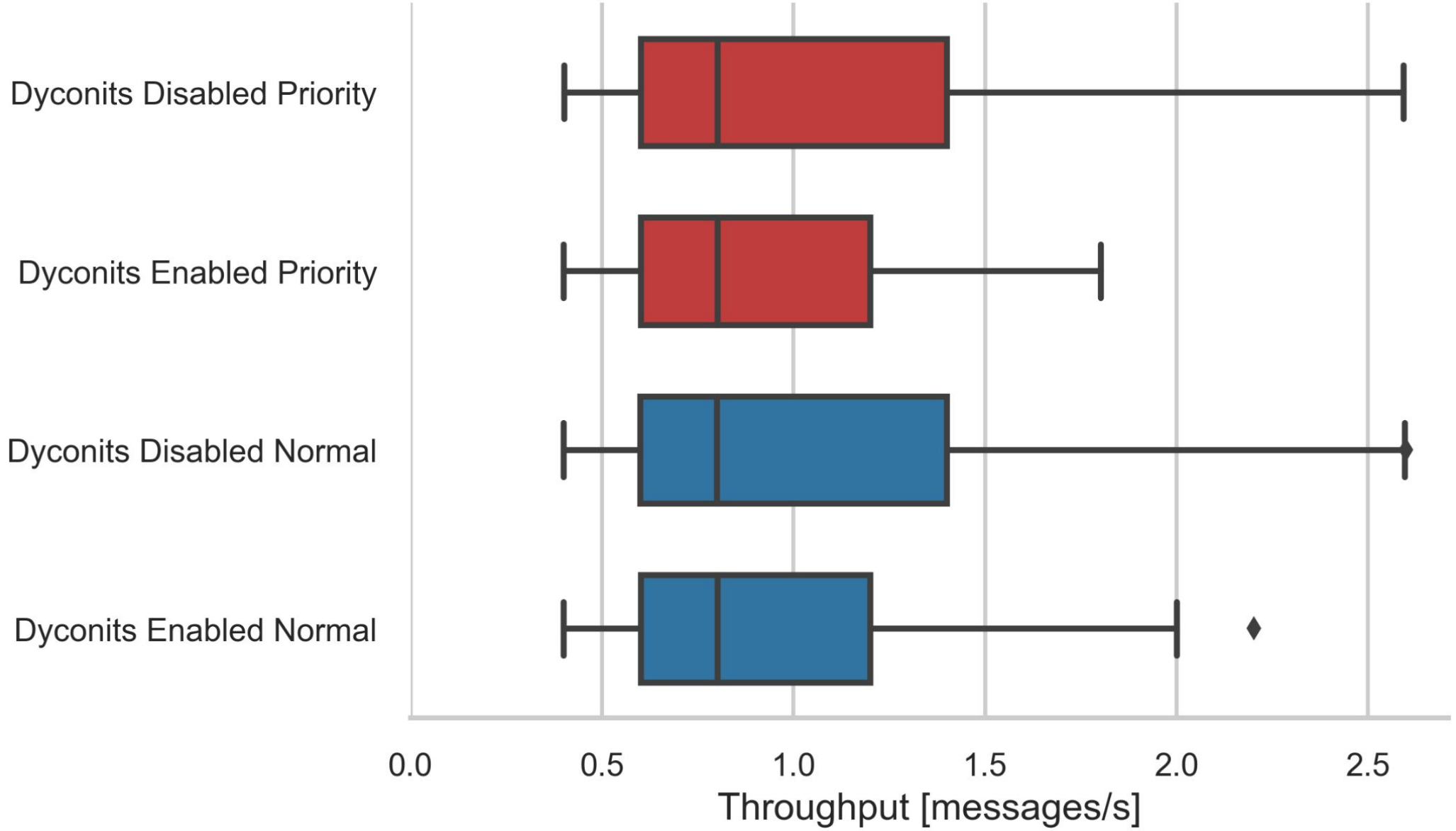
MF9 – The choice of policy affects Hestia's behaviour significantly.

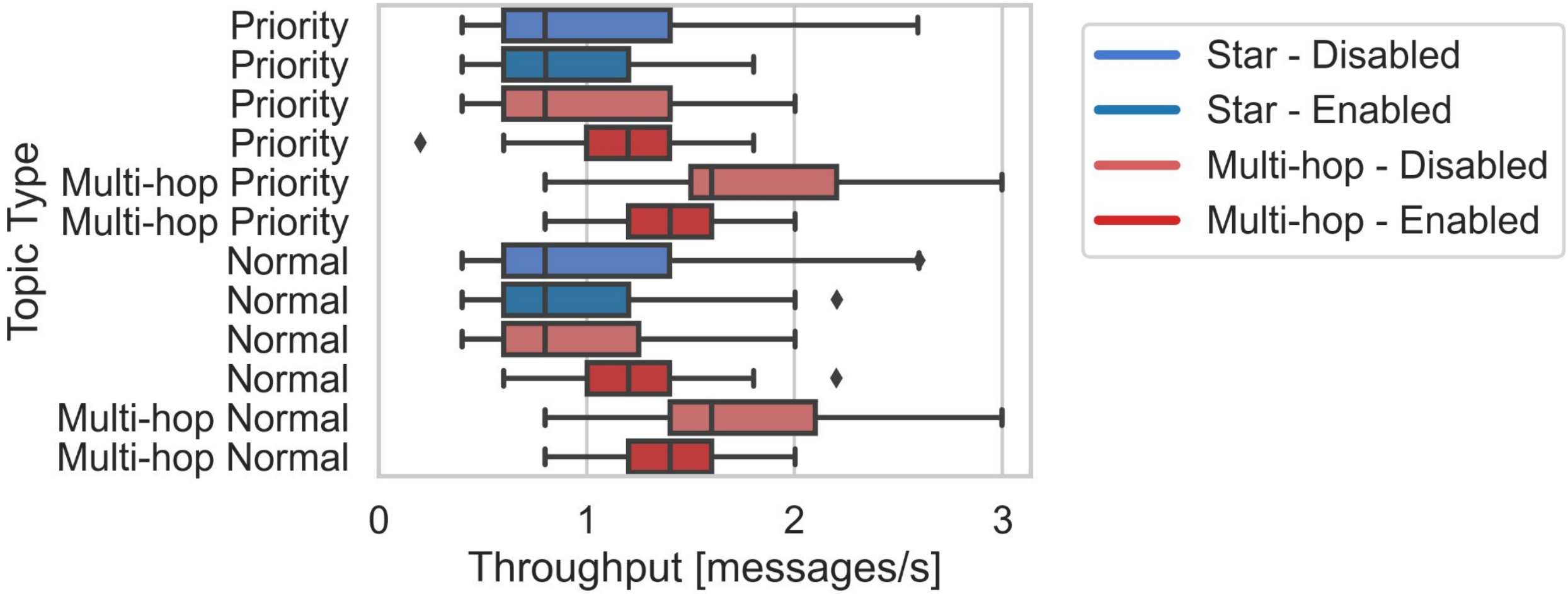


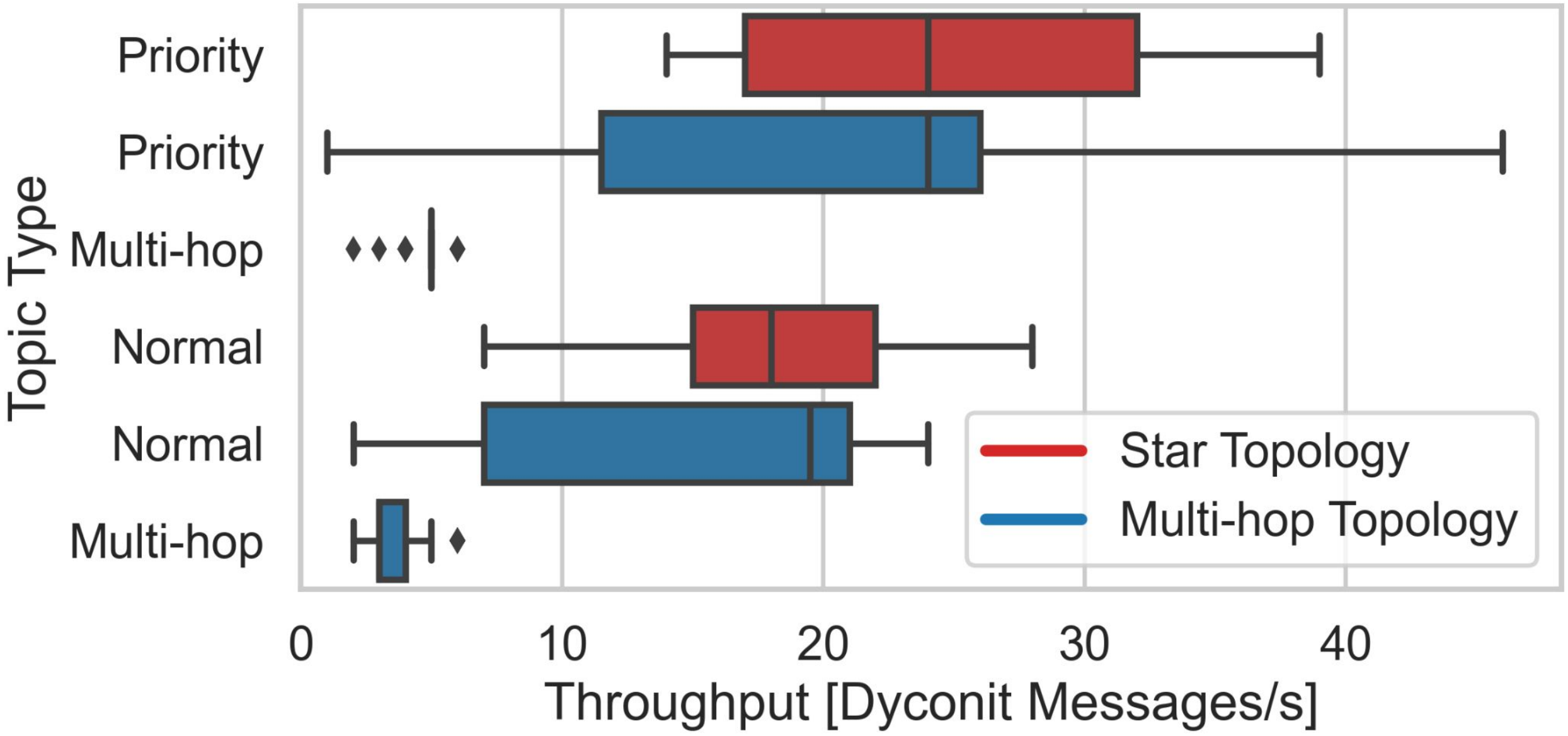


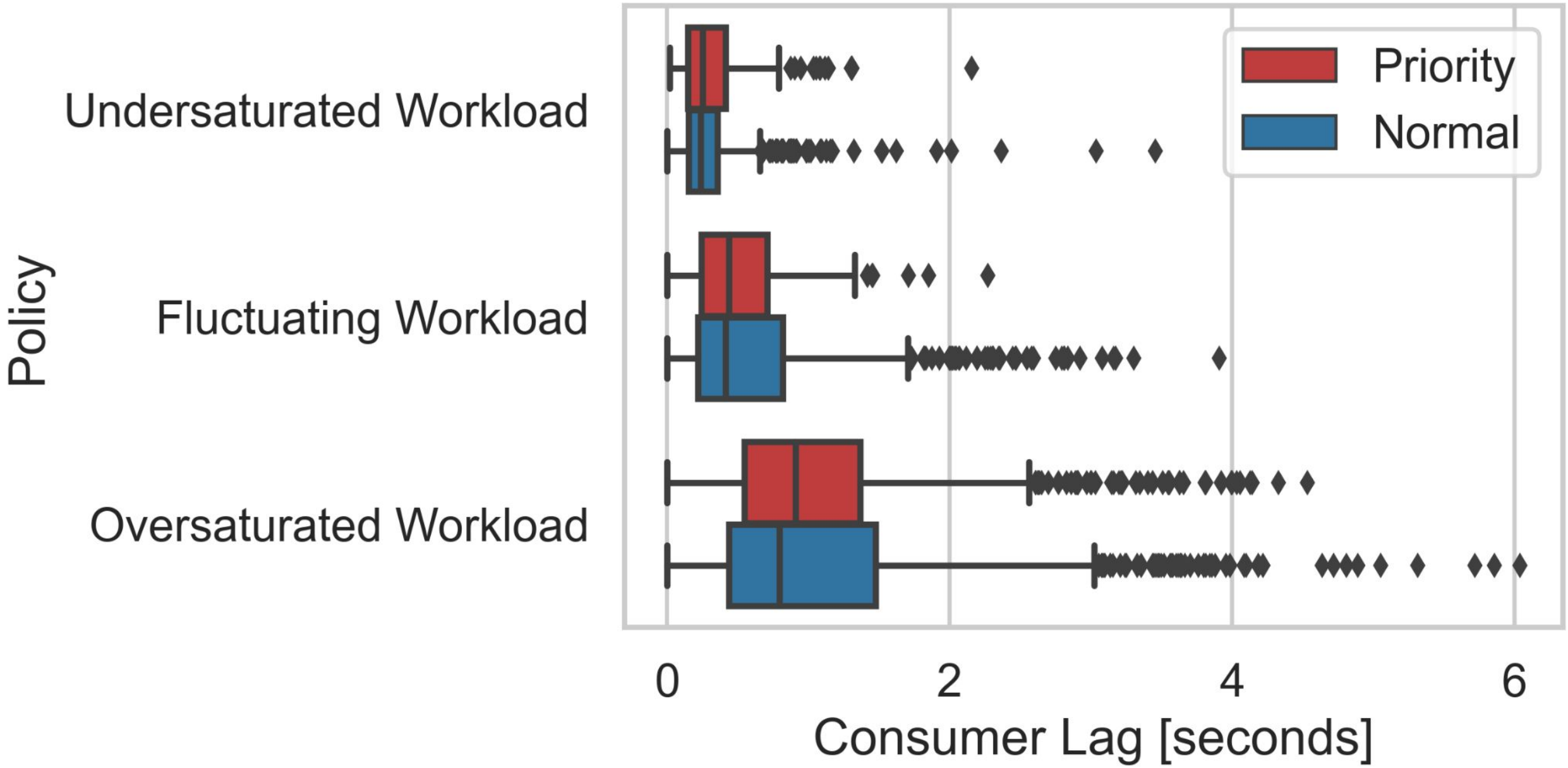

Consumer Lag [seconds]
Dyconits Disabled - priority events
Dyconits Enabled - priority events
0
2
4
6
0
25
50
75
100
125
150
175
200
Consumer Lag [seconds]
Dyconits Disabled - normal events
Dyconits Enabled - normal events
0
2
4
6
8
0
100
200
300
400
500
600
700
800
Events Consumed

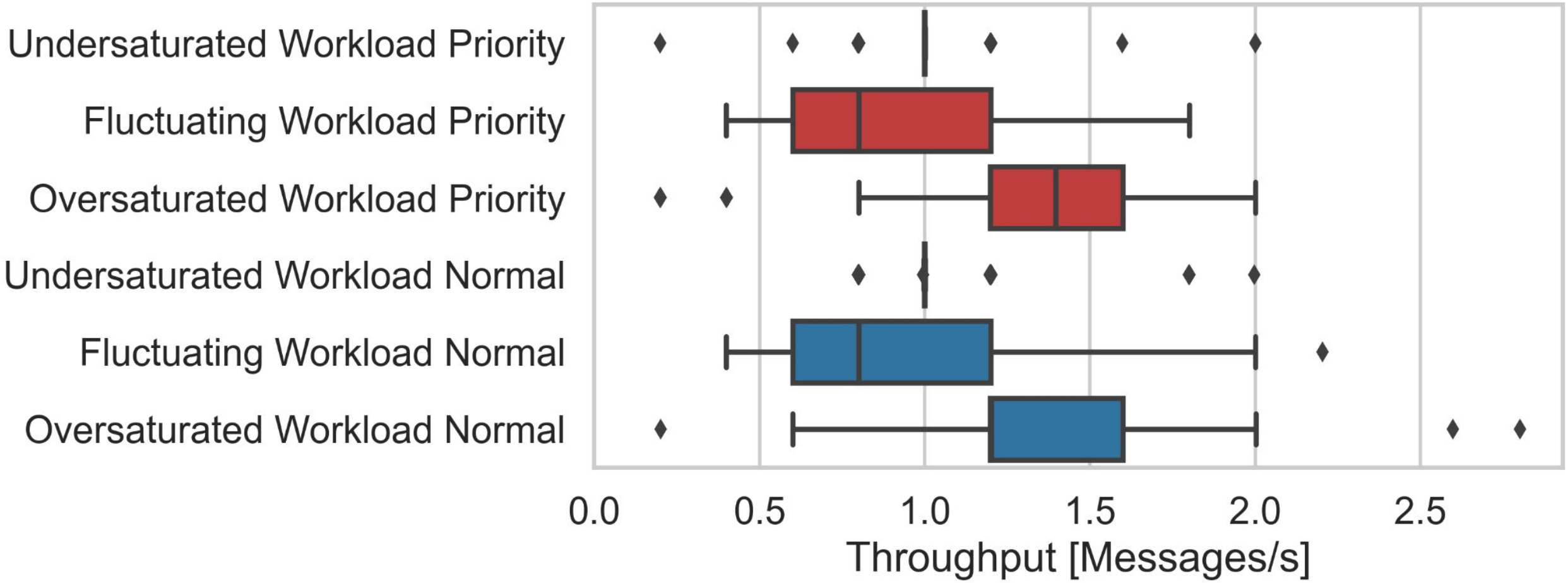

Average Message Throughput / second
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200
Time [seconds]
Priority Events (Dyconits Disabled)
Priority Events (Dyconits Enabled)
Normal Events (Dyconits Disabled)
Normal Events (Dyconits Enabled)

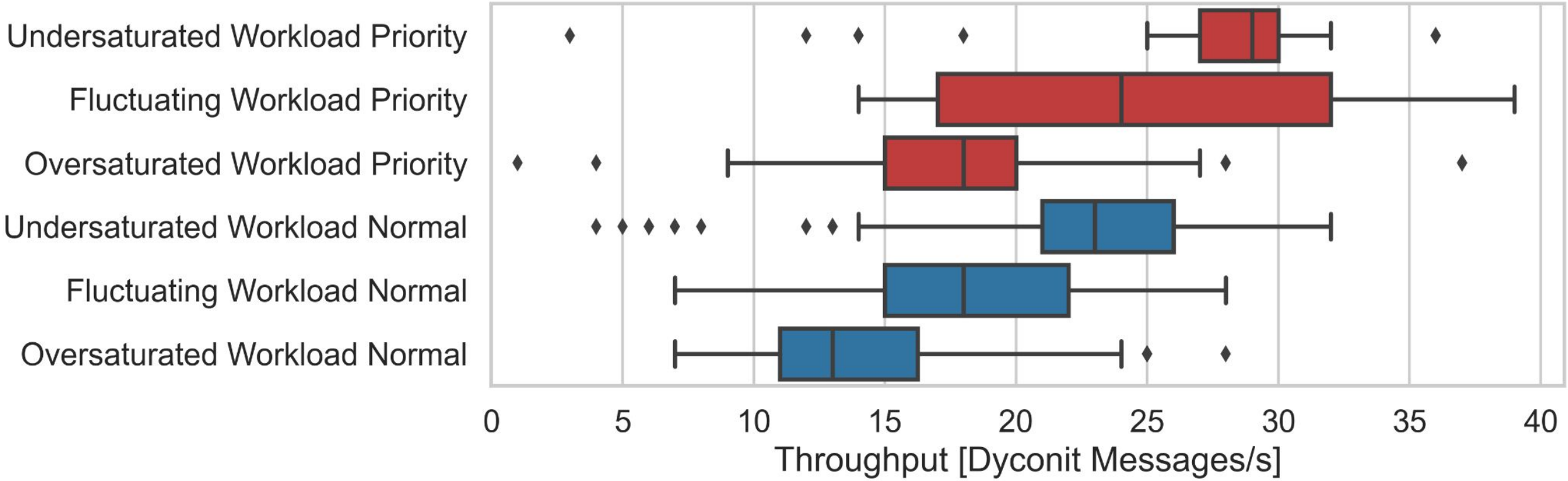

Dyconits Disabled Priority
Dyconits Enabled Priority
Dyconits Disabled Normal
Dyconits Enabled Normal
0.0
0.5
1.0
1.5
2.0
2.5
Throughput [messages/s]

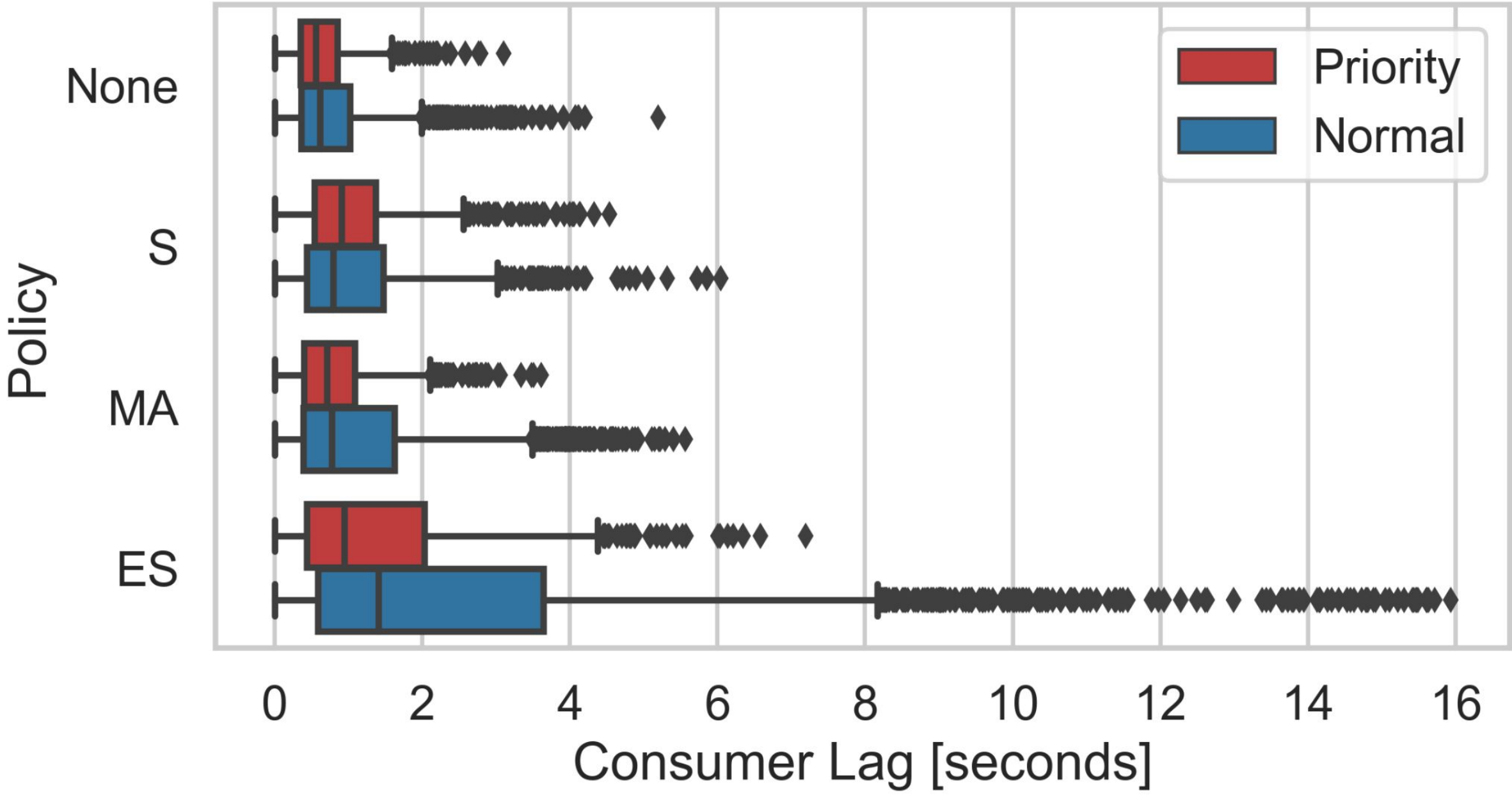

Topic Type
Priority
Priority
Priority
Priority
Multi-hop Priority
Multi-hop Priority
Normal
Normal
Normal
Normal
Multi-hop Normal
Multi-hop Normal
0
1
2
3
Throughput [messages/s]
Star - Disabled
Star - Enabled
Multi-hop - Disabled
Multi-hop - Enabled

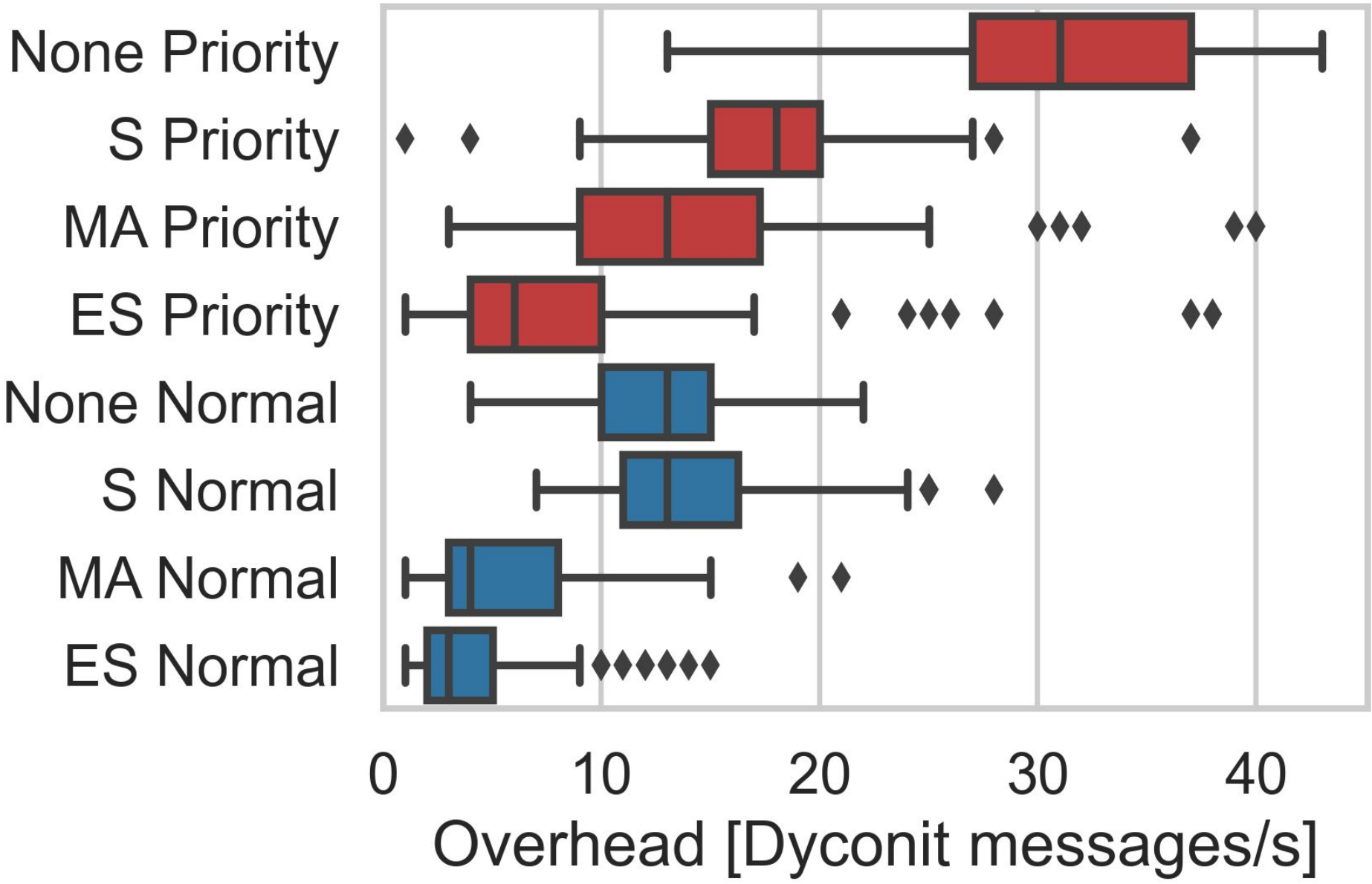

Priority
Priority
Multi-hop
Normal
Normal
Multi-hop
Topic Type
Star Topology
Multi-hop Topology
0
10
20
30
40
Throughput [Dyconit Messages/s]

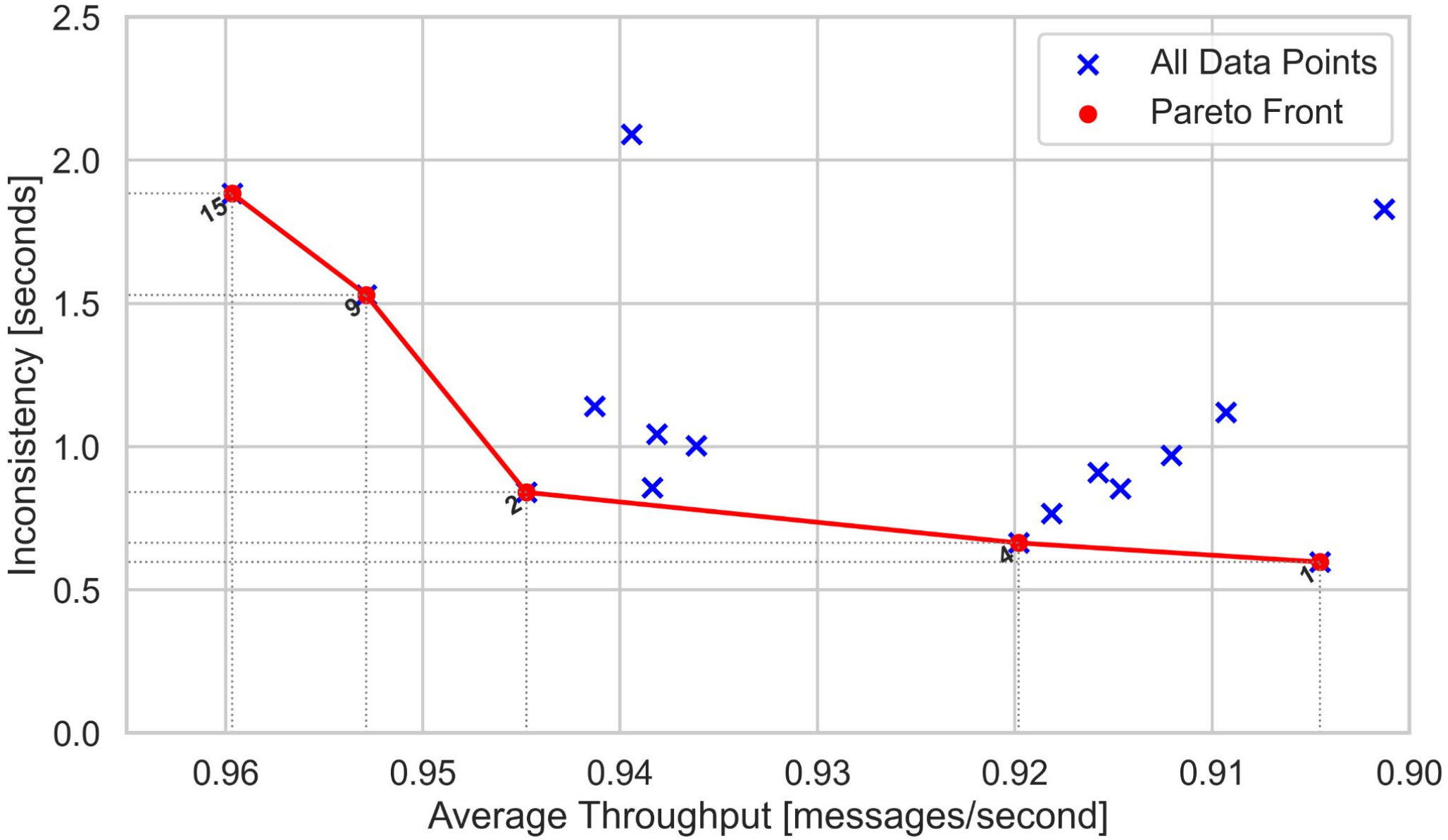

Undersaturated Workload
Fluctuating Workload
Oversaturated Workload
Policy
Priority
Normal
0
2
4
6
Consumer Lag [seconds]

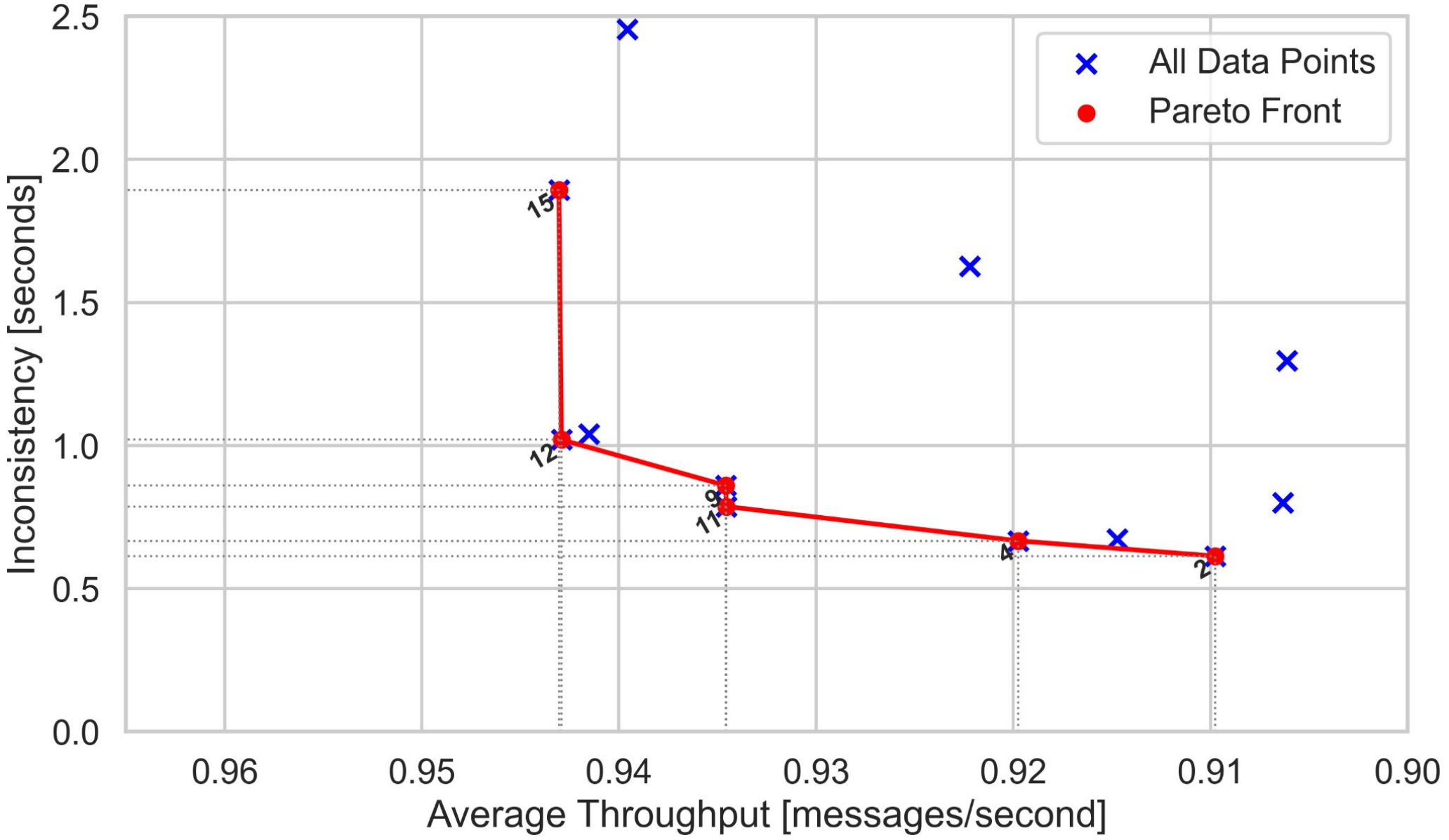

Undersaturated Workload Priority
Fluctuating Workload Priority
Oversaturated Workload Priority
Undersaturated Workload Normal
Fluctuating Workload Normal
Oversaturated Workload Normal
0.0
0.5
1.0
1.5
2.0
2.5
Throughput [Messages/s]

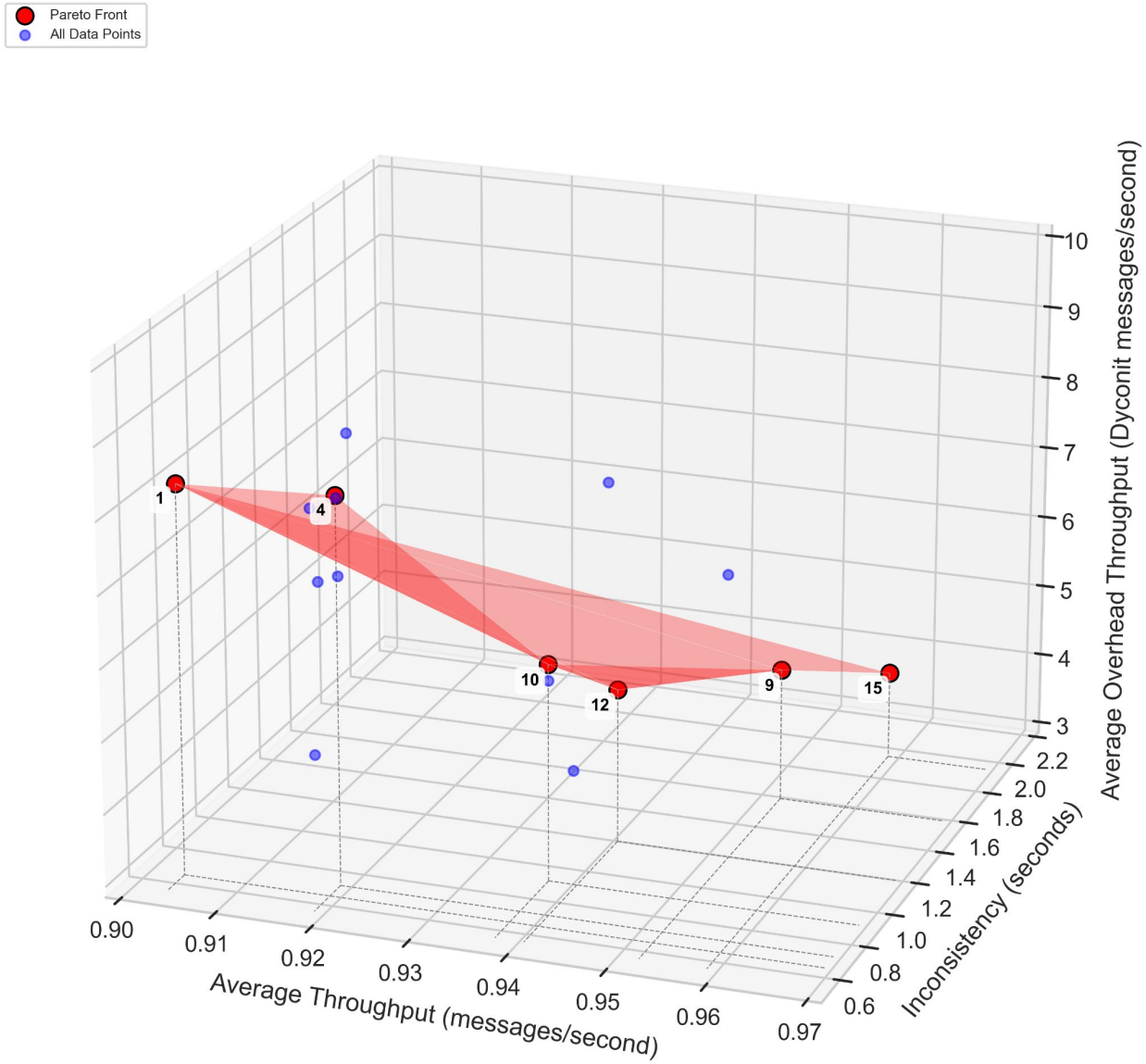

Undersaturated Workload Priority
Fluctuating Workload Priority
Oversaturated Workload Priority
Undersaturated Workload Normal
Fluctuating Workload Normal
Oversaturated Workload Normal
0
5
10
15
20
25
30
35
40
Throughput [Dyconit Messages/s]

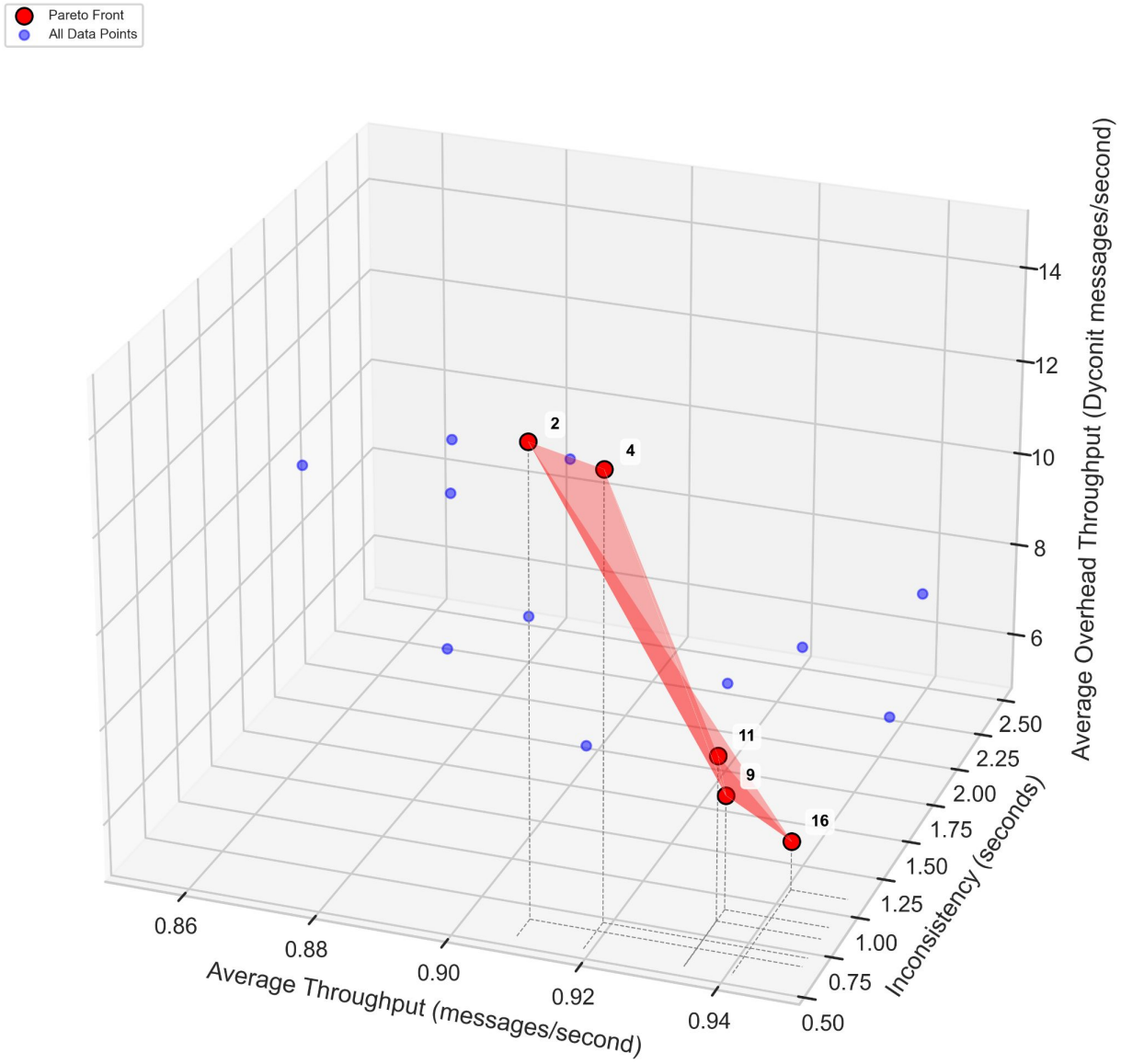

None
S
MA
ES
Policy
Priority
Normal
0
2
4
6
8
10
12
14
16
Consumer Lag [seconds]


None Priority
S Priority
MA Priority
ES Priority
None Normal
S Normal
MA Normal
ES Normal
0
10
20
30
40
Overhead [Dyconit messages/s]


All Data Points
Pareto Front
Inconsistency [seconds]
Average Throughput [messages/second]
2.5
2.0
1.5
1.0
0.5
0.0
0.96
0.95
0.94
0.93
0.92
0.91
0.90
15
9
2
4
1


All Data Points
Pareto Front
Inconsistency [seconds]
Average Throughput [messages/second]
2.5
2.0
1.5
1.0
0.5
0.0
0.96
0.95
0.94
0.93
0.92
0.91
0.90
15
12
11
9
4
2


Pareto Front
All Data Points
1
4
10
12
9
15
0.90
0.91
0.92
0.93
0.94
0.95
0.96
0.97
Average Throughput (messages/second)
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
3
4
5
6
7
8
9
10
Inconsistency (seconds)
Average Overhead Throughput (Dyconit messages/second)


Pareto Front
All Data Points
2
4
11
9
16
0.86
0.88
0.90
0.92
0.94
Average Throughput (messages/second)
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50
Inconsistency (seconds)
6
8
10
12
14
Average Overhead Throughput (Dyconit messages/second)