Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

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

Author: Jurre Joost Brandsen BSc (2724088/11808918)

1st supervisor:	Alexandru Iosup	
daily supervisor:	Jesse Donkervliet	
daily supervisor:	Rinse van Hees	(Info Support)
2nd reader:	Animesh Trivedi	

A thesis submitted in fulfilment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 28, 2023

Abstract

Distributed systems enable online services and applications to scale and distribute workloads. However, they face the challenge of providing consistency, the illusion of sequential access, in a distributed environment. Consistency is essential for reliable and predictable distributed systems that can handle concurrent data access and updates. However, consistency guarantees are often imposed by the underlying system, regardless of the application's specific needs. Application-centric consistency is a novel perspective on consistency models that allows applications to define their own consistency policies based on their requirements and the system's state. This perspective enables fine-grained dynamic consistency in software systems composed of loosely coupled components that communicate through events, which offer benefits such as scalability, performance, and flexibility, but also pose challenges related to consistency, reliability, and complexity.

This thesis extends the concept of dynamic consistency, which was introduced by Dyconits, to a broader range of applications. Dyconits are a novel technique for managing inconsistency in Modifiable Virtual Environments (MVEs), where users can modify the shared state of the system. However, the question of their adaptability to various other domains remains largely unexplored. Contexts as diverse as smart farming, chat rooms, and real-time analysis applications, which necessitate synchronization of replicated data in the face of lagging services, present intriguing challenges that Dyconits could potentially tackle. Consequently, the focus of this thesis shifts towards investigating how Dyconits can be harnessed within event-driven systems to enable optimistic inconsistency across a spectrum of event types and scenarios.

We present Hestia, a dyconit system that enables fine-grained control over the consistency levels of normal and priority events in event-driven systems. Unlike existing approaches that rely on system-level or data-level consistency, Hestia supports application-centric consistency, which allows the application logic and

semantics to define the consistency requirements. We are the first to design, implement and evaluate a dyconit system for general event-driven systems. Hestia employs two general communication patterns that can accommodate various application contexts and domains, such as smart farming, chat application, and real-time analysis. We evaluate Hestia on different communication patterns, workloads and policies and compare it with baseline systems. Our results demonstrate that Hestia achieves a 70% reduction in inconsistency for both event priorities, while maintaining a reasonable throughput with a 45% decrease. Hestia also adapts to the workload saturation by balancing performance and consistency. Furthermore, Hestia enables users to balance consistency, throughput, and overhead according to their preferences and workload characteristics.

Our research highlights the importance of applying application-centric consistency that meets the specific requirements of event-driven distributed systems, recognising the diversity of applications and their varying requirements. We contribute to a comprehensive approach for achieving application-centric consistency in a wide range of contexts.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

List of Algorithms

LIST OF ALGORITHMS

1

Introduction

Distributed systems are essential for modern computing, as they allow systems to scale and distribute workloads across multiple nodes. These systems support various online services and applications that require high scalability and availability, such as e-commerce and social networking. However, designing and implementing distributed systems that provide the necessary levels of reliability, availability, and scalability while maintaining consistency is a significant challenge (1). In particular, achieving *consistency*, the illusion of sequential access, in a distributed environment is a complex task. The problem becomes more challenging as the number of nodes and the frequency of updates increase (2).

Consistency is a key property of distributed systems that ensures the accuracy, reliability, and accessibility of the data stored and managed by these systems. However, achieving consistency is not easy, as distributed systems face various challenges such as network delays and partial failures. To cope with these challenges, different consistency models have been proposed, each with different trade-offs between consistency and availability (3). *Consistency models* are a set of rules that govern the behaviour of a distributed system and define the guarantees provided by the system about the order and visibility of operations on shared data. Some of the well-known consistency models are strong consistency and eventual consistency. Strong consistency models, such as linearizability (4), require that all nodes in the system agree on the order and outcome of operations, and that reads always return the most recent version of the data. However, strong consistency models are often hard to achieve in practice, as they impose high performance and availability costs. Moreover, some theoretical impossibility results limit the design space of distributed systems. For example, the Fischer-Lynch-Paterson result (5) shows that consensus is unattainable in a fully asynchronous message-passing distributed system with even a single crash failure

possibility. Similarly, the CAP theorem (6) proves that a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance. Therefore, many applications and scenarios need weaker or more flexible consistency models that can tolerate some inconsistency in exchange for higher availability or lower latency. As a result, the notion of consistency has been weakened and blurred over the years by the emergence of numerous theoretical boundaries, leading to an explosion of different consistency models (3). New consistency models relax some requirements of strong consistency models, such as atomicity or order of operations, while preserving some intuitive properties or causal dependencies among operations.

One domain that demands both high consistency and high performance in distributed systems is the online gaming industry. This domain faces unique challenges in balancing the trade-offs between consistency, latency, and performance (7). Most existing approaches use data-centric consistency models that focus on synchronising replicas and ordering operations (8). However, these models have drawbacks in their ability to measure and limit the inconsistencies that may arise. To overcome these drawbacks, Donkervliet et al. introduced dynamic consistency units (Dyconits) (9), a middleware for Modifiable Virtual Environments (MVEs) that uses a fine-grained continuous consistency model. This model is based on a definition of consistency that was first proposed in TACT (10), which uses various boundaries to enforce consistency bounds among replicas. Dyconits allow game developers to divide the game world and its objects into units, each with its own bounds, which can be dynamically and optimally adjusted by the middleware. Dyconits enable games to scale by bounding inconsistency in MVEs, while reusing the existing game codebase and network stack. The evaluation of Dyconits showed that they can support 40% more concurrent players and reduce bandwidth up to 85% with only minor code adjustments to the game (9).

Dyconits have gained prominence as effective tools within the online gaming industry, demonstrating their capacity to tackle consistency challenges in complex virtual environments. However, the implications of Dyconits extend far beyond gaming, showing promise in resolving issues of consistency in a multitude of other domains. The foundational principles of dynamic consistency units offer potential applications in various distributed systems where maintaining a delicate balance between consistency, latency, and performance is crucial. Consider real-time use cases like chat applications, Internet of Things (IoT) applications, and data analytics, which could all benefit from Dyconits' capability to dynamically

manage inconsistency and ensure synchronised updates. By generalizing the Dyconits concept, the fine-grained continuous consistency model can be adapted to diverse distributed systems by tailoring boundaries and metrics to suit specific application demands. Further investigation is imperative to evaluate the viability and efficacy of Dyconits in these contexts. Additionally, an exploration of potential trade-offs and challenges arising from their application beyond the gaming industry is essential to comprehensively understand their scope.

1.1 Problem Statement

Distributed systems have become increasingly popular due to their ability to scale and distribute workloads across multiple nodes. However, achieving consistency in a distributed environment is a challenging problem. Traditional consistency models, such as strong or eventual consistency, may not be optimal for all use cases. In recent years, researchers have proposed new approaches, such as Dyconits, that provide a fine-grained solution for dynamically and optimistically ensuring consistency while meeting specific performance requirements (11).

One of the areas where fine-grained dynamic consistency can be applied is in software systems that consist of loosely coupled components that communicate through events. These are messages that represent a significant change in state and enable high scalability, performance, and responsiveness, as well as decoupling and flexibility among components. However, such systems also pose challenges in terms of consistency, reliability, and complexity. Especially in the scenario when multiple consumers are subscribed to the same event. Fine-grained dynamic consistency is a promising approach for addressing the consistency challenges in this context. Its ability to dynamically and optimistically ensure consistency while meeting specific performance requirements makes it an area worth exploring in research to evaluate its effectiveness.

Dyconits have been specifically developed for managing complex interactions within MVEs. Their use in other domains remains unexplored. In this research, we set out to address two primary objectives:

1. **Assessing Dyconits' versatility:** Our aim is to understand the potential of Dyconits in contexts beyond gaming. This will involve studying any inherent limitations and gauging their adaptability to different scenarios.

2. **Exploring optimal utilisation strategies:** We will investigate how to best harness Dyconits to ensure consistency for various types of priority events, taking into account different topologies, policies, and workloads.

Building on the exploration of Dyconits' versatility and optimal utilization, our research delves deeper into the challenges posed by event-driven systems. We propose to build a system that uses the Dyconit consistency model for event-driven scenarios. This system seeks to address the challenges in such contexts by introducing a flexible consistency model tailored to adapt to diverse use cases.

To accomplish our objectives, we will design and develop a Dyconits library with core functionalities that can be easily extended to meet the specific needs of users. The library will be designed for use in asynchronous event-driven distributed systems, which are widely used in industry. We will evaluate the effectiveness of our Dyconits library by testing its performance in a variety of use cases. Ultimately, we aim to provide a reliable and consistent approach to managing interactions between systems, improving the overall efficiency and effectiveness of distributed systems in a range of industries.

1.2 Research Questions

In this work, we aim to generalize the concept of Dyconits and apply it to other domains where event-driven systems are prevalent, such as Internet of Things (IoT), real-time analytics processing, and chat applications. We hypothesize that Dyconits can provide benefits in terms of consistency and performance for these domains as well. To test this hypothesis, we propose the following research questions:

(RQ1) State-of-the-art: What are the common requirements in event-driven systems that can inform the design of a generic Dyconit system?
Understanding the common requirements in event-driven systems is crucial, as it provides valuable insights into the fundamental elements needed for successful system design and implementation. This research question is challenging due to the diverse nature of event-driven systems, encompassing different communication patterns and use cases. Identifying the common requirements necessitates comprehensive analysis and synthesis, considering the trade-offs between generality and efficiency in designing a generic Dyconit system that can accommodate the unique characteristics of event-driven systems.

(RQ2) Design of the system: How to design a generic Dyconit system for event-driven systems?

Research into the use of Dyconits in MVEs has yielded favourable results. These results encourage exploration of the possibilities of extending the application of Dyconits to other domains where scalability and consistency are also of critical concern. However, designing a generic Dyconit system for various application domains poses significant challenges. This is because the concept of Dyconits is new and further research is needed to fully understand its potential for application in various domains. Moreover, the design of a generic Dyconit system must take into account the different requirements and needs in different domains.

(RQ3) Implementation of the system: How to integrate a generic Dyconit system into event-driven systems?

The integration of a general Dyconits into distributed systems is crucial, as it can extend the applicability of Dyconits across a wide range of domains, enabling greater scalability and consistency while maintaining high performance. However, the task of integrating such a system can be challenging because distributed systems are complex and vary widely in their architecture and functionality.

(RQ4) Evaluation of the system: How to evaluate a generic dyconit system for event-driven system?

Optimistic inconsistency lets the system handle temporary data mismatches until they are fixed. This can improve the performance of the system by reducing the latency and overhead of sending messages, as well as increasing the throughput and scalability of the system. However, implementing optimistic inconsistency also poses some challenges, such as how to evaluate the system under different conditions and scenarios, how to configure the system with different policies, workloads, and topologies, and how to understand the trade-offs and behaviour of the system. This research tackles these challenges with experiments and analysis using the Generic Dyconit System prototype and relevant metrics and NFRs.

By ensuring that Dyconits can be used in areas besides MVEs, this work has the potential to directly impact on the choices companies make regarding existing consistency models.

1.3 Main Contributions

By addressing the research questions presented above, we make the following contributions (Cs).

C1 Analysis of three different use cases within the domain of agriculture, finance and instant messaging, where we identified the shared requirements inherent in a particular subset of event-driven systems: replica consumers. This analysis serves as the basis for developing the overall Dyconit system for event-driven systems, specifically tailored to meet these requirements. (Section 3)

C2 Designed a general Dyconit system that can be integrated across various domains, such as agriculture, finance, and instant messaging. This system is the first to extend the Dyconit consistency model. It also addresses the challenge of replica consumer divergence, which is a common issue in event-driven systems. (Section 4).

C3 Development of the Hestia prototype system. This system leverages the Dyconit consistency model to introduce an optimistic inconsistency strategy designed specifically for the challenge of replica consumer divergence within event-driven systems. Hestia enables the use of customisable consistency boundaries and policies, extending the scope of the Dyconit consistency model to a wide range of domains not previously explored (Section 5).

C4 Evaluation of Hestia showing the trade-offs of embracing optimistic inconsistency within the context of event-driven systems characterised by differences between replica consumers. Our experiments show that Hestia successfully mitigates inconsistency through dyconit configuration, while traditional approaches cause replicas to diverge under fluctuating workloads (Section 6).

1.4 Structure of the Thesis

This thesis is organized as follows: Chapter 2 provides the necessary background of the events that motivated the development of the generic Dyconit system for event-driven systems. It analyses the evolution of consistency models, introduces the concept of application-centric consistency, and discusses Conits and Dyconits as well as their alternatives. Chapter 3 analyses the requirements of event-driven systems and presents three use-cases of applications that adopt such systems, identifying common non-functional requirements for

these applications. Chapter 4 describes the design of the generic Dyconit system. Chapter 5 explains the implementation details of the prototype, Hestia. Chapter 6 evaluates the performance and effectiveness of Hestia. Finally, Chapter 7 summarises the main contributions of this thesis and suggests directions for future work based on this thesis.

2

Background

In this chapter, we explore consistency in distributed systems, from its evolution to the application-centric perspective (Section 2.1). We introduce Conits, a model that manages inconsistency in terms of staleness, order, and error (Section 2.2). Finally, we delve into Dyconits, an extension of Conits, which allow applications to dynamically adjust the consistency level based on their current context and preferences (Section 2.3).

2.1 Introduction to Consistency

The word "consistency" is etymologically derived from the Latin word "consistentia", meaning "standing together", and it initially referred to the firmness of matter (12). Later, it evolved to signify agreement, steady adherence to principles, and harmonious connection between the parts of a system. *Consistency* with respect to data means that a consistent state requires that all relationships between data items and replicas to be in agreement and accurately reflect the intended state (13). This definition of data consistency aligns with the original meaning of the Latin word "consistentia", signifying *agreement* as the emphasis is placed on the correctness of the data.

Consistency in distributed systems has been researched for many years and has become a large field of interest. The research into consistency can be divided into four distinct historical periods: the pre-consistency era, the traditional consistency era, the optimistic consistency era, and the tunable consistency era (11). The pre-consistency era was primarily characterized by an emphasis on shared memory architectures and multi-core processors, while later eras such as the optimistic and tunable consistency eras are global in scope.

This section provides a brief overview of each historical period, along with the corresponding well-known consistency models, to create a context for the remainder of this thesis. However, for a more in-depth exploration of consistency models, the reader may consult one of the many surveys on consistency models and mechanisms, as this section only gives a brief overview (3, 13, 14).

2.1.1 Consistency Models in Distributed Systems

Research on consistency in distributed systems has resulted in many consistency models. The large number of consistency models in distributed systems can be attributed to important theories like the CAP theorem, PACELC and ACID properties. The CAP theorem (15) states that a distributed system cannot possibly guarantee all of Consistency, Availability, and Partition Tolerance simultaneously. This theorem states that in the presence of a network partition, the system must choose between maintaining consistency or availability. Therefore, different consistency models offer different levels of trade-offs between consistency and availability. Moreover, the PACELC theorem (16) extends the CAP theorem by introducing the concepts of latency and consistency tradeoffs. The PACELC theorem states that for partitioning, a system must choose between: 1) availability over consistency with low latency (PA/EL), 2) consistency over availability with low latency (PC/EC), 3) availability over consistency with eventual consistency (PA/EC), or 4) consistency over availability with eventual consistency (PC/EL). Finally, ACID (atomicity, consistency, isolation, durability) is a set of properties that ensure reliable processing of database transactions (17). While ACID properties are highly desirable in distributed systems, they often come at the expense of availability and partition tolerance.

The availability of different consistency models allows distributed systems to make the right trade-offs based on their specific usage and requirements. Therefore, understanding these trade-offs and choosing the appropriate consistency model is crucial for designing and implementing efficient distributed systems.

2.1.2 Evolutional Progression of Consistency Models

Distributed systems have evolved over time to offer different levels of consistency, depending on the needs and preferences of the applications that use them. Figure 2.1 shows the four main stages of this evolution: pre-consistency, traditional consistency, optimistic consistency, and tunable consistency. Each stage has its own characteristics and challenges,

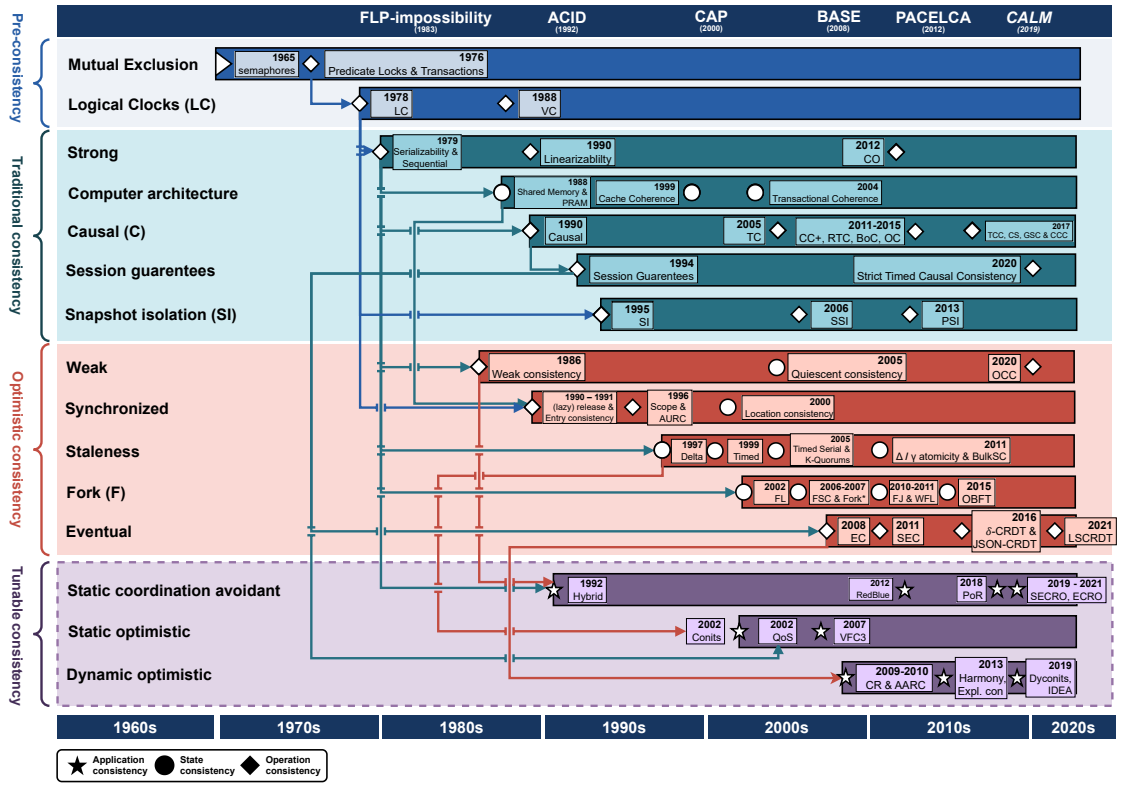


Figure 2.1: A Historical progression of consistency models includes the pre-consistency era (blue) with a focus on mutual exclusion and ordering, the traditional era (green) emphasizing strong consistency, the optimistic era (red) valuing high availability with relaxed consistency, and the tunable era (purple) balancing user-defined consistency needs.

as well as advantages and disadvantages. In general, the trend has been towards more flexible and adaptable consistency models that can balance performance, availability, and correctness. In the following sections, we will describe each of these stages in more detail and discuss their implications for distributed systems design and implementation.

- **Pre-consistency Era:** This era dealt with the problems of mutual exclusion, clock synchronization, and event ordering in distributed systems without a central authority (18, 19). It introduced the concepts of Logical Clock (LC) and Vector Clock (VC), which detected conflicts between data replicas and laid the foundation for models based on causal operations or states.

- **Traditional Consistency Era:** This era aimed to achieve strong consistency, which ensured that all data replicas remained identical at all times. It utilized the mechanisms from the previous era, such as vector clocks, to implement consistency models.

For example, causal consistency ensured that all replicas observed operations in a causal order (20). However, these models sacrificed availability and performance for correctness, resulting in high coordination costs among replicas.

- **Optimistic Consistency Era:** This era relaxed the consistency requirements for better availability and performance. It recognized that some applications could tolerate temporary inconsistencies as long as they were eventually resolved. It proposed ideas like eventual consistency (21), which guaranteed that all replicas would converge to the same state if no new updates occurred. While these models provide more flexibility and scalability, they also introduce challenges, such as resolving conflicts or reconciling divergent states. Therefore, designing a consistency model requires a trade-off between consistency, availability, and performance (22, 23), depending on the specific requirements of the application.

- **Tunable Consistency Era:** This era shifted the focus from universal data consistency to application-specific consistency. It allowed different levels of consistency depending on the application's needs and user preferences. It designed models like RedBlue (24) consistency or CRDTs (25), which took into account factors such as operation commutativity, invariant maintenance, and user perception. This era prioritized adaptability and scalability over strict consistency, emphasizing the unique requirements of each application.

Overall, the tunable consistency era marked a significant shift in distributed systems research, from a one-size-fits-all approach to more fine-grained consistency models based on application semantics and characteristics. This shift enables more flexible, scalable, and adaptable systems that can better meet the specific requirements of different applications.

2.1.3 Application-Centric Consistency Models

Traditionally, consistency models have been considered from two different perspectives (8). The most intuitive perspective is data-centric consistency. *Data-centric consistency* is a model that focuses on maintaining consistency in the data itself. In this approach, all nodes in the system access the same data, and updates or changes to the data are immediately propagated to all other nodes. This ensures that all nodes have a consistent view of the data at all times. Examples of data-centric consistency models include strong consistency, where all nodes see the same data at the same time, and eventual consistency, where updates may take some time to propagate to all nodes.

Client-centric consistency, on the other hand, focuses on ensuring that clients always have a consistent view of the data, regardless of the actual state of the data on each node. In this approach, each client maintains its own view of the data, and any updates or changes are sent to all nodes. This ensures that each client sees the same version of the data, regardless of any inconsistencies that may exist on individual nodes. Examples of client-centric consistency models include read-your-writes consistency, where a client always sees its own updates immediately, and monotonic read consistency, where a client never sees older versions of the data after seeing a newer version (6).

Recently, we introduced a third perspective, known as *application-centric consistency*. Unlike the previous two perspectives, this model focuses on the requirements of the specific application rather than the data or the clients (11). In other words, the consistency guarantees are tailored to the application's needs rather than being determined by the underlying system or network. This approach enables greater flexibility and performance optimizations, as well as more nuanced consistency guarantees that can better balance trade-offs between consistency and other factors such as availability and performance. However, it also introduces greater complexity and requires more careful consideration of the specific application requirements.

Figure 2.2 illustrates the differences between the three consistency perspectives described above. To illustrate what we mean by application consistency, let us consider a hypothetical scenario of a software developer who is designing a distributed system for a ride-sharing application. The developer faces a trade-off between ensuring that the data is consistent across all nodes in the system, and achieving high performance and availability. Using traditional consistency models that focus on the consistency of the whole system might result in poor performance or availability. Alternatively, the developer could use an application-centric consistency model that adapts to the specific needs of the ride-sharing application. For instance, the developer could prioritize strong consistency for critical operations such as matching riders with drivers or calculating fares, ensuring that these operations take no more than a second to complete, while allowing for weaker consistency for less critical operations such as displaying ride history, which can take several seconds up to a minute. By doing so, the developer can optimize performance, availability, and scalability for the ride-sharing application.

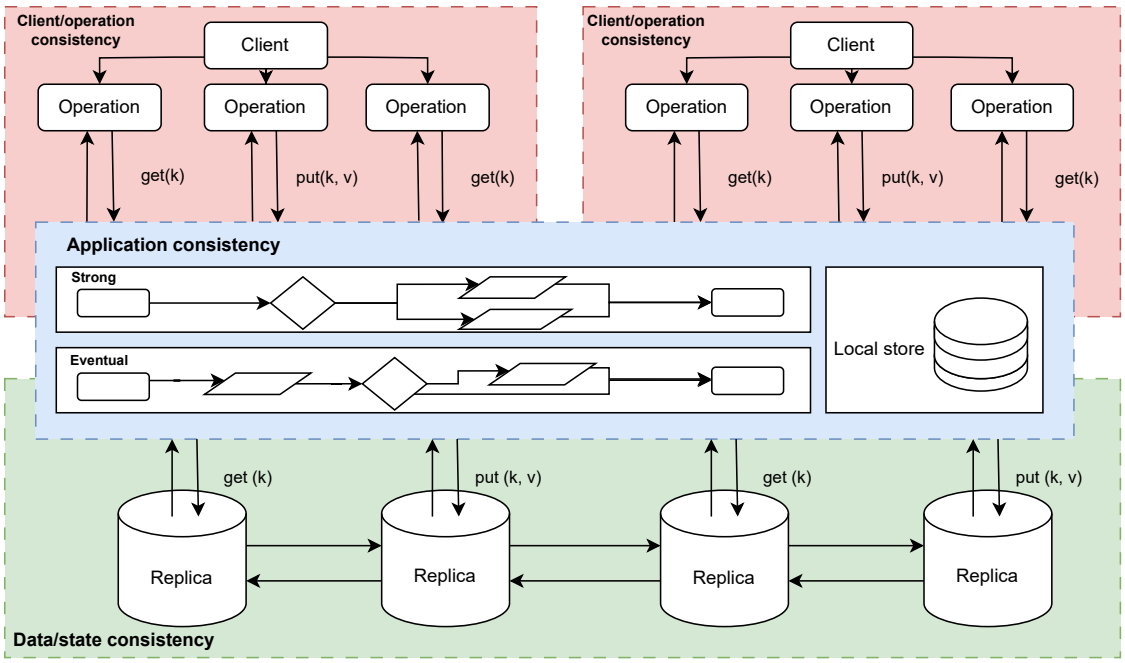


Figure 2.2: The difference between data-centric (green), client-centric (red) and the novel application-centric (blue) perspectives of consistency models in distributed systems.

The application-centric consistency perspective distinguishes four subcategories: static, dynamic, coordination avoidant and optimistic.

- **The static subcategory:** In this category of consistency models, software engineers specify the level of consistency for each operation or a group of operations (i.e., by using bounds) in advance, without the possibility of changing said bounds or the consistency level during the execution of the application.

- **The Adaptive subcategory:** This category focuses on models that are designed to allow consistency models to change dynamically based on various factors. These factors may include user preferences, policies, or conditions such as network latency or data availability.

- **The Optimistic subcategory:** Consistency models in this category avoid unnecessary synchronisation overhead by allowing for temporary inconsistencies within set bounds or probabilities. This trade-off gives more flexibility and performance optimization. Optimistic consistency is useful in systems that need high scalability and availability, but can allow some temporary inconsistency.

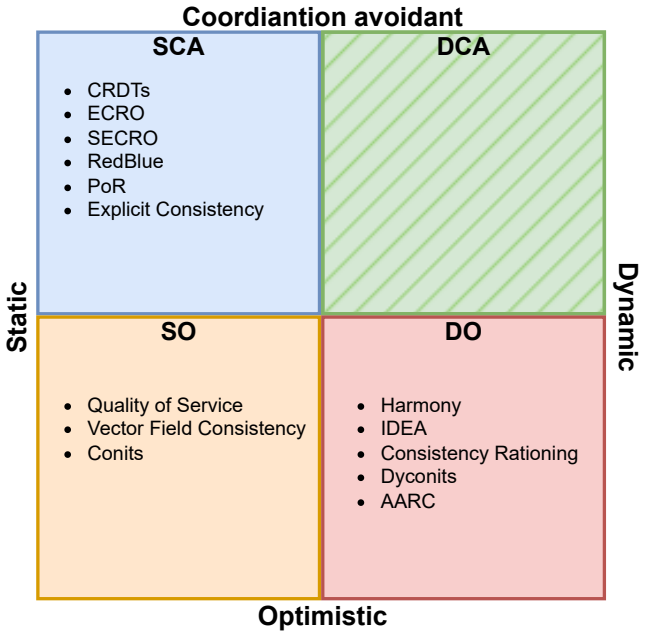


Figure 2.3: A taxonomy of application-centric consistency models based on their static or dynamic nature and their coordination avoidant or optimistic behaviour.

- **The Conflict-free subcategory:** This subcategory uses invariant or coordination-free mechanisms to avoid conflicts and minimize communication between nodes. Occasionally, it may need limited coordination for a few operations to resolve conflicts. Coordination avoidant consistency is relatively inflexible in terms of runtime flexibility because the data structures and algorithms have the consistency guarantees built into them. This leaves little room for adjustments or changes during execution. The software engineer still needs to know and choose the right data structure and algorithm that are consistent with the application logic.

In Figure 2.3, the taxonomy of application-centric consistency models is shown. In this taxonomy, the subcategories 'static' and 'dynamic' and 'coordination avoidant' and 'optimistic' are orthogonal to each other. This is because we have observed that the models previously classified as 'novel' consistency models can be categorized as either static or dynamic, and either coordination avoidant or optimistic. This taxonomy provides a useful framework for comparing and contrasting the different approaches to consistency from the application-centric perspective, emphasizing their distinctive features and commonalities.

2.2 The Conit Consistency Model

This section introduces the Conit consistency model (10), which is a key concept for this thesis. The thesis proposes a generalised version of the Dynamic Conit consistency model, which is based on an adaptation of the original conit model called Dyconit. To appreciate the novelty and benefits of the proposed model, it is essential to first review the basic principles of the Conit model.

2.2.1 Continuous Consistency

In 2002, Yu and Vahdat (10) proposed a novel approach to defining inconsistencies using three axes: numerical error, staleness, and order error, which form continuous consistency ranges. To quantify and bound inconsistency between nodes, the Conit consistency model allows systems to define an arbitrary number of integer bounds in the domain $[0, \infty)$. Every operation, whether read or write, indicates if it affects or depends on one or multiple Conits. Inconsistency is quantified over updates, and writes are also called updates. The Conit model allows consistency to vary between the extremes of linearizability and eventual consistency. The Conit consistency model guarantees that the inconsistency of the data presented to the user never exceeds the configured value. If the bounds are exceeded, synchronisation between nodes is required before the operation can be completed. Synchronisation guarantees that the system consistency stays within bounds from the user's perspective and involves communication of all updates between nodes, but no updates are discarded. The system is fully consistent when all nodes have seen all updates.

In terms of specifying inconsistency tolerance, applications can use different approaches. For example, measuring inconsistency in terms of numerical deviations is useful for data with numerical semantics, such as a stock market application. Numerical deviation can also be understood in terms of the number of updates that have been applied to a given replica but have not yet been seen by others. Staleness deviations relate to the last time a replica was updated. In some applications, ordering deviations are allowed, as long as the differences remain bounded. Tentatively applying updates to a local copy and awaiting global agreement from all replicas before making them permanent is one way to handle ordering deviations.

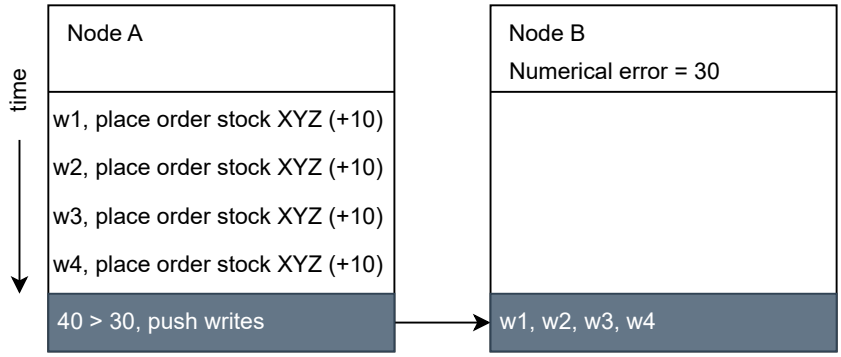


Figure 2.4: Illustration of the effect of setting a numerical error bound in a Conit

2.2.2 The Notion of a Conit

Yu and Vahdat (10) proposed a way of defining inconsistencies through the concept of a *consistency unit*, which they call a *conit*. A conit serves as the unit for measuring consistency and its definition is flexible, allowing programmers to define it based on their needs. Examples of Conits can range from shares in a stock exchange to seats on an airplane.

To gain a more comprehensive understanding of the impact of each of the three metrics used in a conit, an illustrative case is explored, involving the application of Conits within a hypothetical stock market. We assume that the stock market can detect numerical deviations, staleness deviations, and order errors and can specify the maximum acceptable range of inconsistency for each metric. In this example, a Conit is shared between two nodes, A and B. Both nodes maintain a log of Conits, where each conit represents a contiguous segment of the database containing the latest prices of stocks.

2.2.2.1 Numerical Error

The numerical error of a node is the sum of the weights of writes that it has not seen, adjusted by a global factor. Nodes share their numerical error bounds with others and propagate writes to keep the error below the bound. The propagation decision is based on local information only. Figure 2.4 shows an example of setting a numerical error bound. Two database replicas log transactions from a stock market application. Node B sets a bound of 30 stocks, meaning that the unseen writes cannot exceed 30 shares in total. A client submits buy orders for 10 shares of a stock to node A. Each order adds a weight of 10. The first three orders are allowed without communication. The fourth order exceeds the bound, so node A must send its writes to node B before accepting more orders.

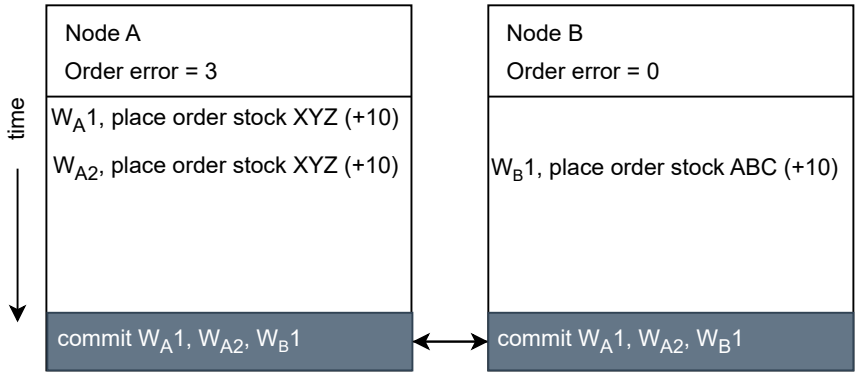


Figure 2.5: Illustration of the effect of setting an order error bound in a Conit

2.2.2.2 Order Error

The order error of a node is the number of writes that can be reordered without affecting the data consistency. Nodes set their order error bounds and commit writes to agree on their order when the bounds are exceeded. The system can use any algorithm to commit writes. Figure 2.5 shows an example of setting an order error bound. Two replicas log transactions from a stock market application. Node A sets a bound of 3 and node B sets a bound of 0. Node A receives an order for 10 shares of ABC, which increases its order error to 1. Then, node A and B receive two orders at the same time. Node A gets 10 shares of ABC, while node B gets 10 shares of XYZ. Node A's order error becomes 2, while node B's order error becomes 1. This violates node B's bound, so it must communicate with node A to commit the writes and establish a total order. After the commitment, the order errors are reset to zero.

2.2.2.3 Staleness Error

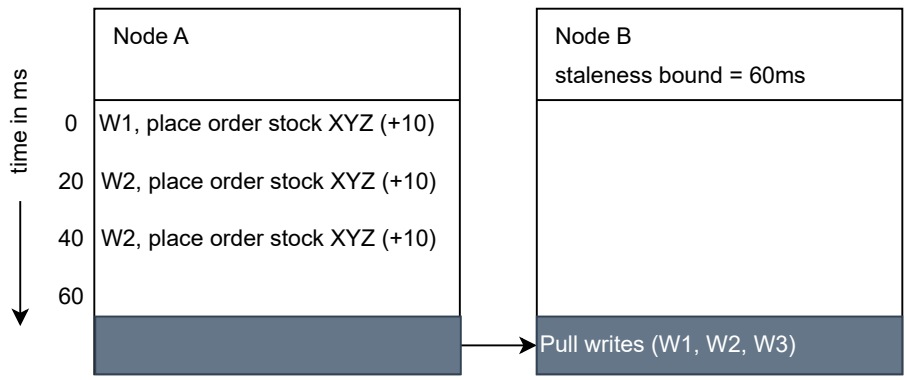


Figure 2.6: Illustration of the effect of setting a staleness bound in a Conit

The staleness error of a node is the time difference between its latest update and the most recent update in the Conit. Nodes set their staleness bounds and pull updates from other nodes when the bounds are reached. The Conit model allows the system to choose any method to pull updates. The original paper used a pull-based approach that blocked the node until it received the updates. Figure 2.6 shows an example of setting a staleness bound. The vertical axis is time in milliseconds. Node B sets a bound of 60 ms, meaning that it must pull updates from node A if it has not seen any for 60 ms. Node A receives a write every 20 ms. After the third write, node B's bound is exceeded, so it initiates the pull process.

2.3 Dyconits

The Conit consistency model, which we discussed in the previous section, defines inconsistencies along three axes. However, this model has some limitations when applied to large distributed systems, as Donkervliet (26) pointed out. Specifically, the Conit model does not handle nodes joining and leaving the system dynamically, assumes that clients can only update state speculatively, ignores the heterogeneity of nodes, and lacks scalability. To overcome these challenges, Donkervliet et al. introduced Dyconits (9), a modified version of the Conit model that addresses the identified issues.

Donkervliet's team used Minecraft-like games, known as Modifiable Virtual Environments (MVEs), as a vehicle to research optimistic bounded inconsistency. MVEs are particularly relevant for this research because the gaming industry is a major contributor to the development and advancement of large-scale distributed systems (27, 28, 29).

Dyconits is a novel model for managing inconsistencies in large-scale distributed systems, which introduces the concept of dynamic Conits, known as *dyconits*. These are consistency units that represent arbitrary subsets of the game state. It tackles the challenges encountered by the original Conit consistency model by serving as an intermediary layer between the game-code and networking layers of the MVEs. Specifically, the Dyconits middleware employs optimistically bounded inconsistency, which optimizes the consistency of the system and the efficiency of its operations. The outcome of deploying the Dyconits middleware is an increase in the number of concurrent players by up to 40% and a reduction in network bandwidth usage by up to 85%, all accomplished through minimal alterations to the pre-existing game-code.

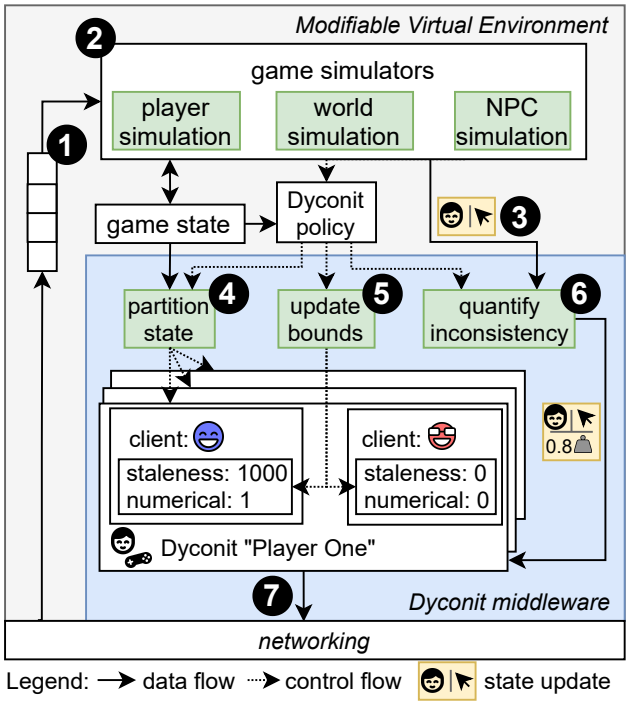


Figure 2.7: Dyconit system architecture. The figure shows how the Dyconit system acts as a middleware between the MVE server and the clients, without requiring any modifications on the client-side. The system dynamically quantifies and bounds inconsistency using dyconits, which are subsets of the virtual world that have different consistency requirements. The system updates the inconsistency bounds based on the changing interests of each client and forwards state updates only when the bounds are exceeded.

2.3.1 Dyconit System Design

In this subsection, we will delve into the system design of Dyconits. The system design is focused on fulfilling five primary requirements. These requirements have been formulated with the goal of addressing the limitations of the Conits consistency model. First, it needs to reduce system-wide network usage to prevent scalability bottlenecks (**R1**). Second, the system needs to quantify and bound optimistically the inconsistency of the virtual environment (**R2**). Third, it needs to allow fine-grained control over system inconsistency to manage different types of user interest (**R3**). Fourth, the consistency bounds need to be modified dynamically to prevent decreasing gameplay experience (**R4**). Finally, the system must be simple yet flexible to act as middleware in distributed ecosystems (**R5**). The fulfilment of **R5** is exemplified by Figure 2.7, which depicts the middleware capabilities and flexibility of the Dyconit system in distributed ecosystems. The Dyconit middleware, shown in blue in the figure, acts as a mediator between the MVEs existing networking and

simulation layers, without any modifications on the client-side. The dyconit system interacts only with the MVE server and dynamically quantifies and bounds inconsistency using dyconits. Incoming state updated are directly queued (❶) for processing by the game's simulators (❷). However, the resulting state updates (❸) are no longer sent directly to the clients. Instead, they first pass through the Dyconit system. The system quantifies and bounds inconsistency through the use of dyconits (**R2**). The subsets (i.e., partitions) are created dynamically (❹) by the Dyconit system using the selected policy. Similarly to the Conit concept, each dyconit is capable of bounding the staleness, numerical inconsistency, and order inconsistency to provide inconsistency bounds for player avatars, virtual-world objects, or any combinations of these. The system updates inconsistency bounds dynamically (❺), based on the changing interests of each player. To bound inconsistency, the system quantifies the inconsistency caused by each state update (❻), forwards it to the corresponding dyconit, and evaluates the consistency requirement for each client. If a client's inconsistency bounds are exceeded, the system forwards all state updates to the client (❼).

Network bandwidth consumption is reduced in two ways (**R1**): First, the system merges messages with the same state, allowing for large reductions in bandwidth usage for state that is modified frequently. Second, queuing state-updates allows for batching, reducing system-level overhead such as packet headers.

Bounding the inconsistency is optimistic because the MVE does not reduce availability while synchronising state updates (**R2**). To better understand why bounding the inconsistency is optimistic, let us consider a situation where a client has reached its *staleness limit*. The staleness limit is a threshold that determines how long a client can go without updating its information. If the client's information is not updated, it may become inconsistent. However, updating the information takes time due to network delays and processing delays, which can cause the time between sending the message to the system and it being processed by the client to exceed the staleness limit. To avoid this issue, the system can set the staleness limit to the maximum latency that the client can tolerate and deduct the estimated network and processing delays.

To allow for fine-grained control over the inconsistency (**R3**), the Dyconits system manages each dyconit dynamically and automatically. Furthermore, dyconits are re-configured dynamically (**R4**) to match consistency restrictions that benefit each player. To achieve this, the Dyconit systems allow programmers to specify *policies* (**R5**). Policies are sets

of rules that define consistency requirements for each dyconit. The Dyconits middleware then enforces these policies to dynamically adjust dyconit configurations based on player needs. Each Dyconit policy can affect the operation of Dyconits in three areas. First, the policy determines how the global state is partitioned across dyconits. Second, the policy determines the weight of individual state-updates; accumulation of weight beyond the bounds leads to synchronisation. Third, the policy can re-configure dyconit bounds for each player, based on player state and the dynamic system workload.

2.3.2 Adaptability, Extensibility, and Generality of Dyconits

In this thesis, we explore the use of the Dyconit system, a fine-grained consistency model for distributed systems, and evaluate its performance and adaptability under various scenarios. The Dyconit system is an instance where the application-centric consistency perspective is employed, offering a nuanced approach that is dependent on environmental factors or particular applications that may offer other types of consistency. As such, it is part of a small group of consistency models that have adopted this perspective. However, the field is seeing progress in this area, as some recent works have advocated for the creation of systems that can adjust service level objectives (SLOs) as resources overload or tasks lag (30). The Dyconit system demonstrates the feasibility of this approach by adopting policies that adapt to real-world circumstances, which conceptually support the vision of adapting SLOs to changing circumstances. Moreover, Donkervliet et al. argue that this system's mechanisms and architecture could be extended to other domains, although they leave this conjecture to future work (9). This presents an opportunity to explore the applicability of the Dyconit system to different domains and evaluate its effectiveness. We aim to investigate the feasibility and performance of using the Dyconit system in a distributed system for a specific domain under various conditions. Our study seeks to provide insights into the potential of this system's policies and mechanisms to support fine-grained consistency and adaptability in distributed systems beyond gaming

3

Requirements Analysis for Dyconit Systems in Event-Driven Systems

This chapter analyses the requirements for event-driven systems, addressing (RQ1). The chapter is organised as follows: Section 3.1 introduces event-driven systems, explains the basics of message brokering and pub/sub systems, and focuses on a specific message broker called Kafka. Section 3.2 describes the methodology for the requirement analysis that is conducted in this chapter. Section 3.3 discusses three use cases of applications that use consumer replicas and determines their non-functional requirements. Section 3.4 presents two communication patterns that together represent the general patterns used in event-driven systems. Finally, Section 3.5 summarises our analysis and discusses how Dyconits can be used as a solution to the challenges presented in the use cases.

3.1 Introduction to Event-Driven Systems

In the ever-evolving landscape of software architecture, one paradigm has gained significant traction for its flexibility, scalability, and responsiveness: the event-driven architecture (EDA). At its core, an EDA is a design pattern in which software components execute actions in response to events. These events are typically significant changes in state or external triggers that the system must acknowledge and act upon (31). There are several key characteristics and advantages of event-driven systems (32, 33):

- **Enable Real-time Responses:** Since systems react immediately to events, user interactions become more fluid, and data processing is more timely.

- **Promote Scalability:** Components can be scaled independently based on event loads, making it easier to manage resources and costs.
- **Enhance Flexibility:** New components or functionalities can be added with minimal disruptions, as they can simply plug into the existing event stream.
- **Improve Resilience:** Since components operate independently, failures in one part often don't ripple through the entire system.

Although the universe of event-driven architectures is vast and varied, our focus in this thesis will be quite specific: we are particularly interested in systems that employ multiple consumer replicas for a single topic. These replicas not only consume the same event data, but can also share the processed work among themselves. This configuration, although incredibly powerful, poses its own challenges, especially when consistency is a priority. This intriguing interplay of efficiency and consistency in such systems will be the cornerstone of our next discussions. To this end, in the following sections, we will delve deeper into the mechanics, tools and real-world applications of event-driven systems, always keeping in mind our primary focus on multi-replica consumers.

3.1.1 Basics of Message Brokering and Pub-Sub Systems

In this subsection, we will introduce two central concepts that event-driven systems rely on. These are message brokering and publish-subscribe (or Pub-Sub) models. *Message Brokering* is about mediating communication between different components of a system – these can be different components of one system or entirely different systems. The key element here is the message broker, a special entity whose primary role is to receive messages from producers, decide where they should go and make sure they get to the right consumers. For example, in an online shopping system, a message broker could route messages from the order service to the payment service, the inventory service, and the delivery service, based on some criteria, such as topic, queue, or priority.

In the *Pub-Sub model*, the roles are clearly defined. Entities, called *publishers*, produce messages. These messages are not sent directly to specific recipients. Instead, they are published to a central hub. On the other hand, we have *subscribers* who express interest in specific types of messages. The central hub, which assumes the role of message broker, ensures that messages from publishers reach all interested subscribers. This way, event-driven systems can achieve decoupling, scalability, and responsiveness.

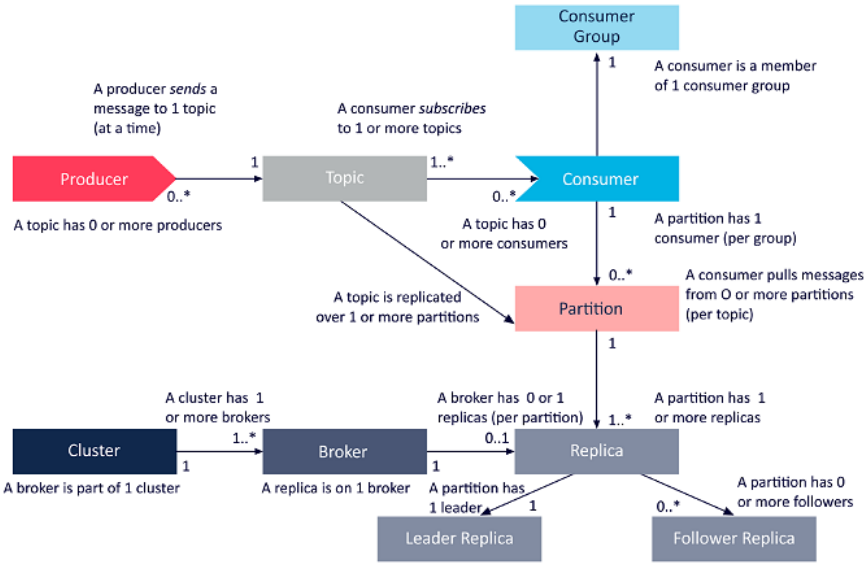


Figure 3.1: Kafka cluster architecture.

The Pub-Sub model is different from other communication models, such as point-to-point or request-response. In *point-to-point communication*, each message has a single sender and a single receiver. In *request-response communication*, each message has a sender who expects a reply from a receiver. In both cases, the sender and the receiver need to know each other's identity and location. In contrast, in the Pub-Sub model, the sender and the receiver are unaware of each other's existence and location. They only communicate through the message broker based on their interests.

There are many popular message brokers and pub-sub platforms that implement these concepts, such as Kafka, RabbitMQ, or Azure Service Bus. These technologies provide various features and options for building reliable and scalable event-driven systems

3.1.2 Introduction to Kafka

The landscape of event-driven systems and real-time data pipelines brims with diverse platforms, each bringing its own set of advantages. While RabbitMQ excels in flexible routing and Azure Service Bus integrates seamlessly with Microsoft services, we've chosen

to explore Kafka for its unique combination of attributes:

- **Industry Trust:** Over 80% of Fortune 100 companies lean on Kafka for their mission-critical data operations (34).
- **Scalability & Efficiency:** Kafka's design emphasizes high scalability, low latency, and formidable throughput, adeptly handling vast data volumes and rapid event rates.
- **Fault Tolerance:** Data replication across Kafka's multiple brokers ensures strong fault tolerance and high availability.
- **Stream Processing:** Kafka's inherent real-time data processing capabilities set it apart in the streaming platform ecosystem.
- **Consumer Groups:** Kafka's approach to parallel event processing, via consumer groups, offers unparalleled efficiency in data handling.
- **Metrics & Monitoring:** Kafka provides exhaustive metrics, such as throughput, latency, errors, and availability, facilitating a nuanced understanding of event states and application performance tuning.

With these compelling attributes, especially for event-driven systems relying on multiple consumer replicas for a single topic, Kafka becomes an evident choice for detailed exploration. Moving forward, we use Figure 3.1 to explore the core components of Kafka in detail.

- **Kafka Cluster and Brokers:** A *Kafka cluster* is a collection of servers that run Kafka, with each server being termed as a *broker*. These brokers are responsible for storing and serving data related to various topics. Their distributed nature ensures Kafka's notable scalability and resilience.
- **Topics and Partitions:** Within Kafka, a *topic* is essentially a stream of events. Topics are further subdivided into *partitions*, allowing the distribution of a topic across multiple brokers. This division aids in parallelizing data reads and writes across various brokers, enhancing throughput and performance. When a new event enters a topic, it's appended to a specific partition. Kafka ensures that events with identical keys (like a customer ID or vehicle ID) go to the same partition. As a result, consumers reading from a partition always see events in the order they were

written (35). To bolster fault tolerance, topics can have multiple replicas based on a defined *replication factor*. In such a setup, one partition takes up the leader role, while the others act as followers. The leader handles all data requests, and in case of failures, one of the followers takes over the leader's responsibilities.

- **Producers and Consumers:** A *producer* in the Kafka ecosystem is an entity or application that sends or publishes events to topics. While producers can specify the partition for an event, Kafka can also assign events automatically, either based on a key or in a round-robin fashion. Conversely, a *consumer* subscribes to topics to process their events. It can start reading from a particular offset or continuously track the most recent events.

- **Consumer Groups:** A unique aspect of Kafka is its *consumer group* concept. A consumer group comprises multiple consumers working in tandem to consume events from a topic. Kafka ensures that each event in a topic is consumed by one and only one consumer within a group. This design ensures both efficient data processing and that no event is overlooked. This partitioning of work among consumers in a group facilitates parallel processing, enhancing throughput. Furthermore, Kafka's decoupled nature means producers and consumers operate independently, a pivotal aspect driving Kafka's scalability.

3.2 Methodology for Requirement Analysis

In this section, we present our approach to analysing the requirements for event-driven architectures. Our methodology, although non-traditional, derives its strength from practical insights. It blends informal discussions, expert consultations, and exploratory thought experiments.

As part of our requirement analysis, we engage in informal discussions with industry experts at Info Support, often over lunch or a cup of coffee. Info Support is the specialist in developing high-quality software solutions and a leader in artificial intelligence (AI), cloud architecture, Managed Services and IT training across various sectors including healthcare, finance, agriculture, food, and retail. Their comprehensive knowledge provides vital insights, especially about diverse EDAs.

During these casual interactions, we explore the fundamental characteristics of EDAs,

discussing both their advantages and challenges. We touch upon the various events underlying these architectures, the quality attributes they possess, and any necessary trade-offs. In addition, we investigate best practices essential for EDA design. As part of our discussions, we delve into real-world applications and scenarios. Notably, we emphasize on applications where multiple duplicate consumers listen to the same topic.

While our methodology might seem unconventional compared to more structured approaches, we design it with a clear purpose. EDAs are dynamic and multidimensional. Often, the insights from practical experiences, such as the ones Info Support provides, surpass what more structured academic methods can offer. Therefore, we ensure our analysis remains grounded in real-world insights.

3.3 Real-World Use Cases of Event-Driven Systems

This section presents three use-cases of applications that employ event-driven systems, illustrating their key features and challenges. The first use-case is a 'smart IoT farming application', which demonstrates how event-driven systems can enable timely and efficient management of critical resources such as water and crops. The second use-case is a 'distributed chat room application', which shows how event-driven systems can facilitate asynchronous communication and state management among multiple participants. The third use-case is a 'real-time analytics application', which highlights how event-driven systems can support business intelligence and decision-making by processing and analysing large volumes of data. These three use-cases exemplify the versatility and potential of event-driven systems, as well as the trade-offs and complexities involved in their design and implementation.

3.3.1 Smart IoT Farming Application

Smart farming, exemplifying an IoT-driven application, utilizes sensors and trackers to oversee and regulate diverse farming facets. Generating multiple events every second, the system is resilient to occasional event losses or delays. For efficient communication and device interaction, an EDA is employed. The sensors, acting as producers, gauge factors such as soil moisture, temperature, humidity, and pH. They generate events based on their measurements and user instructions. On the other hand, the consumers, comprising applications, exhibit the data, offer alerts or suggestions, or modulate irrigation and fertilization systems, reacting to subscribed events.

What makes this use case particularly intriguing for our study is the challenge posed by real-time data integration. A significant problem emerges when essential services encounter delays due to bottlenecks in processing. Let us focus on a scenario centred around the `crop_data` topic within an IoT-integrated farm and examine potential solutions to overcome these bottlenecks. The system consists of services tailored for specific functions, each reliant on distinct data segments. For example, Service A, responsible for irrigation, primarily uses moisture and temperature data to control watering. Service B orchestrates the greenhouse climate by analysing various atmospheric parameters. However, services like Service C, an Analytics and Reporting tool, face a unique challenge. Due to complex data processing tasks or its batch processing nature, it might lag in real-time data assimilation. Typically, this service would independently analyse the data. Yet, its latency can lead to substantial operational postponements, thus becoming a bottleneck. However, Dyconits present a promising solution to this challenge. Services A, B, and C can operate under a shared Dyconit. If Service C lags, either in staleness or numerical order, Services A and B can forward their processed data to it. This enables Service C to promptly integrate these insights into its reports. Despite its processing constraints, this approach ensures Service C remains up-to-date without the immediate need to process raw data.

Given the intricacies of such a smart IoT farming application, one must consider several non-functional requirements. We consider aspects such as performance, reliability, availability, and consistency as key non-functional requirements for this application. This is because the efficiency and effectiveness of smart farming hinge on timely, uninterrupted, and accurate data processing and action. Delayed or inaccurate data can lead to suboptimal farming decisions, potentially affecting crop yields, and jeopardizing the financial viability of the farming operation. A complete list of non-functional requirements for this application is:

- **Performance:** The system should deliver events in near real time with low latency and high throughput.
- **Reliability:** The system should handle event loss or delay gracefully and recover from failures quickly.
- **Availability:** The system should be operational at all times and provide backup or alternative solutions in case of disruptions.

- **Consistency:** The system must ensure the right level of consistency for different types of events. While some events might tolerate inconsistency due to their non-critical nature, others that influence farming decisions directly must be managed with strict consistency measures.

3.3.2 Distributed Chat Room Application

We present a messaging application as the second case study to exemplify distributed applications in EDAs. This distributed messaging service draws inspiration from advanced platforms like Discord[1], Twitch Chat[2], and Microsoft Teams[3].

Our application allows users to create and join chat rooms, send and receive messages, and leave chat spaces. These actions generate events that need to be handled efficiently. To do so, the application uses services of the same type, but with different consumers reading from a common topic. For example, one service filters spam messages, another encrypts and decrypts messages for privacy, and another transcribes voice and media communications. This way, if a consumer in any service delays or fails, another consumer can take over, accessing and processing the shared data, ensuring no service disruption or degradation. This is useful when the application faces high demand and needs to provide a smooth user experience. Imagine a scenario like new-years-eve, where there is a surge of events. The services can ensure that if a consumer is lagging too much, it can catch up with the processed information to maintain consistency.

Different events or data within the application warrant varying levels of consistency. For less crucial events like notifications, analytics, media archiving, and message transcription, eventual consistency suffices. These elements can undergo periodic updates and propagation without disrupting the app's primary functions. Conversely, essential events such as user authentication, profile adjustments, chat room initialization and participation, and message sequencing and dispatch require strong consistency. Furthermore, software engineers can utilize Dyconits to set distinct consistency standards for diverse events, such as messages or notifications from friends versus those from strangers.

[1] `https://www.discord.com/`
[2] `https://www.twitch.com/`
[3] `https://www.microsoft.com/microsoft-teams`

A chat application has several non-functional requirements (NFRs) that it needs to fulfil. These NFRs include scalability, latency, and consistency, which affect the design and operation of the application:

- **Scalability**: The chat application should be capable of accommodating a substantial number of concurrent users and messages.
- **Latency**: The chat application should provide near-real-time communication and interaction between users.
- **Consistency:** The chat application should ensure consistent event propagation and ordering across all participants, offering a synchronised view of events. It is essential to guarantee strong consistency for critical user experience and functionality events, such as authentication, profile settings, chat room creation and joining, as well as message ordering and delivery. However, for less critical events like notifications, analytics, media storage, and message transcription, the application should ensure eventual consistency.

3.3.3 Real-time Analytics

The final use case showcases a 'stream processing' analytics system, designed to analyse data in real-time across multiple nodes. This often requires stream enrichment through methods like database lookups, ETL transformations, or appending machine learning scores (36). In this application, payment gateways generate events based on user actions, such as credit card purchases. Alert services respond to these events, notifying stakeholders of pertinent issues like fraudulent transactions.

To ensure data integrity, pivotal events—like those tied to fraud detection—leverage strong consistency guarantees. However, less critical ones, such as report generation, can rely on eventual consistency. This application uses replicated consumers that read from the same topic. This means that all service instances receive the same events from the topic and process them in the same way. If one consumer falls behind or fails, it can swiftly synchronize by receiving processed data from another active consumer rather than fetching it again from the broker. Thus, if a consumer responsible for appending machine learning scores to events falls behind, it can swiftly synchronize by receiving processed data from another active consumer rather than fetching it again from the broker. This capability

ensures that all parts of the system maintain consistency and resilience, even when individual components encounter hiccups.

To meet the requirements of a real-time analytics application, we adhere to the recommendations provided by Stonebraker et al. (37). Their work continues to play a major role in the development of new systems for real-time analytics (38), despite being almost two decades old. These requirements encompass scalability, latency, and consistency.

3.4 Communication Patterns in Event-Driven Systems

Event-driven systems primarily rely on specific communication patterns to effectively relay information across distributed services. In this section, we focus on the star and multi-hop topologies. We choose these two topologies for two main reasons. First, they offer distinct benefits for EDAs. The star topology, which has a central hub that connects all the other nodes, enables efficient broadcasting and centralized management, which are essential for dynamic and complex system environments. On the other hand, the multi-hop topology, which involves data passing through multiple intermediate nodes before reaching its destination, provides higher resilience and redundancy, which reduce the risk of system failures. Second, we argue that a system that can operate effectively within these two contrasting topologies can also adapt to other, less conventional topological designs. Therefore, our aim is not only to compare the strengths and limitations of the star and multi-hop configurations, but also to propose a versatile system that can handle a wider range of topologies in real-world scenarios.

3.4.1 Star Topology in Event-Driven Systems

The star topology, as shown in Figure 3.2, is a communication pattern that enables publishers to disseminate messages without specifying subscribers. Rather than addressing messages to specific receivers, publishers classify them into categories, denoted as topics in this thesis, and broadcast them to all interested subscribers. Conversely, subscribers indicate their interest in one or more topics and receive only the relevant messages, without being aware of the publishers' identities. The star topology often employs an intermediary entity, called a message broker or event bus, to which publishers post messages and from which subscribers retrieve messages. The broker is responsible for filtering and routing messages from publishers to subscribers according to their subscriptions. Moreover, the broker may queue and prioritize messages before delivering them.

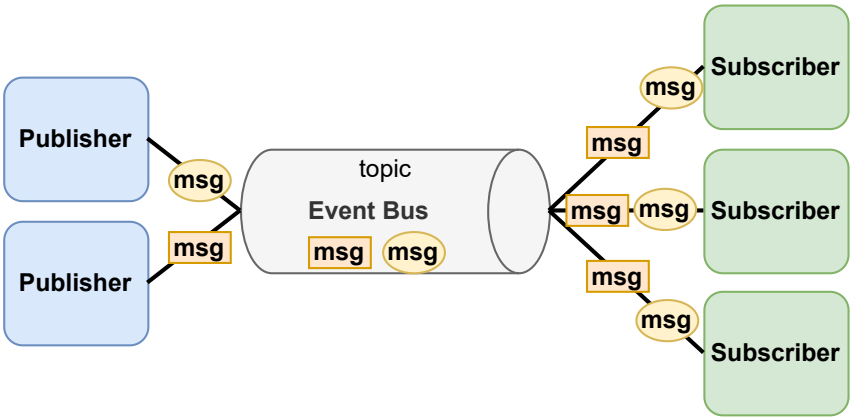


Figure 3.2: The star communication pattern enables message exchange between different services of an application or system. Publishers send messages to a message broker or event bus, which filters and routes them to subscribers according to their subscriptions. Subscribers receive only the messages that match their interests, without knowing the publishers' identities.

To illustrate this communication pattern, consider the chat application from the use cases introduced in the previous section. In our distributed chat application, users generate various events when they send messages, join rooms, or perform other actions. This dynamic can be visualized within the context of a star topology, where the chat application acts as the publisher and the various services, such as the spam filter, encryption, and decryption services, act as subscribers. These subscribers express interest in specific topics or events, ensuring they only receive the relevant messages without needing to know which user or part of the application generated the event. This makes the application highly scalable and ensures that if a consumer in any service delays or fails, another can take over, accessing and processing the shared data, thus ensuring no service disruption.

The star topology offers several benefits (39, 40):

- It enables decoupling of subsystems, which allows for independent management of each subsystem and ensures reliable message delivery even in the presence of offline or unavailable receivers. This enhances the scalability and improves the performance of the sender.

- It increases the reliability, as asynchronous messaging allows applications to cope with increased loads and handle intermittent failures more effectively.

However, the star topology also comes with some drawbacks stemming from its main strength: The decoupling of publisher and subscriber. As the number of subscribers and publishers grows, the increasing volume of messages exchanged may compromise the stability of this communication pattern; it may collapse under high loads.

Some examples of large-scale throughput instability include:

- Load peaks: intervals in which subscriber requests exceed network throughput, followed by intervals of low message volume (underutilised network bandwidth).
- Delays: as more and more applications use the system (even if they communicate on different pub/sub-channels), the message flow to a single subscriber will decrease.

As with all systems, the star topology is not without its challenges. Centralizing communication can create bottlenecks, especially under heavy loads. To address these drawbacks, we propose to use Dyconits as intermediaries between the publisher and the subscriber. Dyconits enable optimistic inconsistency within certain bounds, which means that they can accept some degree of divergence among the services as long as it does not exceed a predefined threshold. Under heavy load, dyconits can tolerate more inconsistencies within a bound, which allows them to cope with the increasing volume of messages and enhances the performance of the system. When the bound is reached, they can either enforce synchronisation among the services and request the publisher to reduce the event production rate, or increase the bounds further to allow for more inconsistency as a trade-off for better performance. Alternatively, under low load of events, dyconits can dynamically adjust the bound using their policies to ensure a higher level of consistency.

3.4.2 Multi-hop Topology in Event-Driven Systems

In this section, we show how Dyconits can be applied to a more complex communication pattern than the star pattern: the multi-hop pattern. Multi-hop communication is a messaging pattern that involves a chain of events triggered by a single message. For example, a publisher ‘A’ sends an event to a message broker, which is retrieved by a subscriber ‘B’. Based on the information received from ‘A’, ‘B’ produces its own event and publishes it to the same or a different message broker. A subscriber ‘C’ listens for this event and receives it from the message broker. Multi-hop communication also inherits the drawbacks of the star communication pattern, such as the lack of feedback and the possibility of overload. Therefore, it is essential in this communication pattern that each point where events are

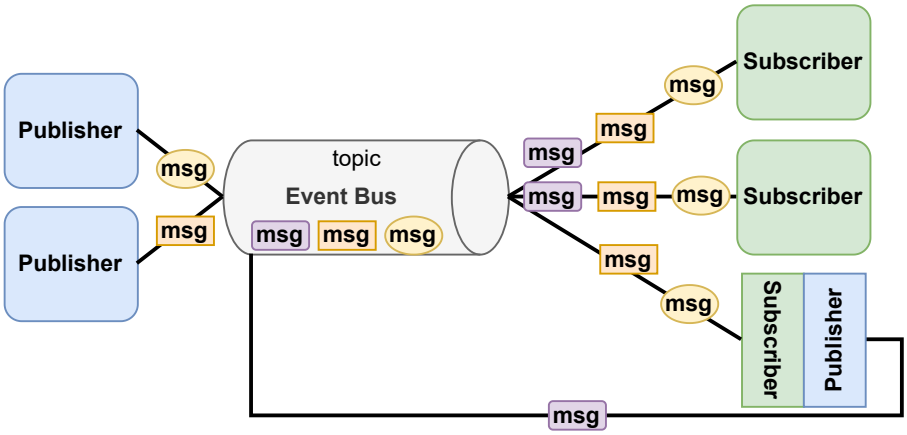


Figure 3.3: Multi-hop communication pattern.

retrieved or published considers all other events generated based on the previous event.

To illustrate this communication pattern, consider the real-time analytics use case described earlier. This application uses a stream processing analytics system where payment gateways generate events based on user actions, such as credit card payments. These events traverse through various nodes, like database lookups, before being enriched with machine learning scores. Once enriched, another service may take this event and generate an alert which is then propagated further, maybe to a stakeholder notifying system. The ability of this system to use replicated consumers that can synchronize data among each other reinforces the multi-hop messaging pattern, with data hopping from one node to another until its final destination. By using Dyconits in this communication pattern, each microservice can focus on its own task without worrying about the events that triggered it or the events that it will trigger because the dyconit consistency model will ensure that when services are out of sync, they will eventually converge to a consistent state. This way, the data quality and reliability of the multi-hop communication system can be maintained.

3.5 Discussion and Implications of Dyconits in Event-Driven Architectures

Event-driven architectures power various real-world applications. Each use-case presented—a smart IoT farming application, a distributed chat room, and real-time analytics—reflects the multifaceted challenges and requirements of implementing EDAs.

However, across the use-cases, we observed challenges such as:

- **The Need for Consistency:** Different events may necessitate varying degrees of consistency. Both priority and non-priority events should be managed to provide an optimal experience without compromising data integrity.
- **High Availability and Reliability:** Systems must be operational at all times, given their real-time nature. Downtimes can have significant repercussions.
- **Managing Bottlenecks:** In event-driven architectures, bottlenecks can manifest in various forms, hindering system performance and compromising the real-time guarantees of the system.

Therefore, we propose that by creating a general Dyconit system for event-driven systems, we can mitigate these challenges by allowing for optimistic inconsistency, which can enhance system performance during peak usage periods while maintaining acceptable bounds on stable periods. This adaptability can enhance the performance and resilience of event-driven systems, especially in scenarios with varying workloads.

In addition, in scenarios with multiple replicated consumers, Dyconits' capabilities are further pronounced. Replicated consumers reading from the same topic ensure resilience. If one consumer lags (i.e., it exceeds either the staleness or numerical error bound), another can promptly synchronise by receiving processed data from an active consumer. Dyconits, adept at managing event flows and processing loads, determine the optimal strategy for when data forwarding based on its bounds. This results in optimistic inconsistent system behaviour and optimised resource usage.

As a result, we advocate for a general Dyconit system design tailored for EDAs. This system would offer:

- **Flexible Consistency Management:** Ensure optimal performance and data integrity by dynamically adjusting consistency based on event nature.
- **Enhanced Resilience:** Marry the strengths of replicated consumers and Dyconits to ensure high system availability and reliability.
- **Real-time Monitoring and Management:** Adapt to event flows, resource allocation, and potential bottlenecks by using Dyconits' inherent adaptability.

We use this enumeration to create a complete list of functional and non-functional requirements for the generic dyconit system for event-driven systems in Chapter 4.

To sum up, while EDAs present unparalleled advantages, they come with inherent challenges. However, with a Dyconit system's integration, these challenges can be navigated effectively, ushering in an era of more robust, efficient, and resilient event-driven solutions.

3. REQUIREMENTS ANALYSIS FOR DYCONIT SYSTEMS IN EVENT-DRIVEN SYSTEMS

4

Design of a Generic Dyconit System for Event-Driven Systems

This chapter presents the design of a generic Dyconit system for Event-Driven Architectures, addressing (RQ2). The chapter is organized as follows: Section 4.1 defines the core functional requirements for the Dyconit system to integrate seamlessly with EDAs. Section 4.2 gives a high-level overview of the Dyconit system design for EDAs. Section 4.3 zooms in on the Dyconit Overlord and the Dyconit Admin components, providing a detailed view of their internal structure and interaction. Section 4.4 explains how the Dyconit system satisfies our functional requirements by bounding consistency using dyconits. Section 4.5 describes the design of dynamic policies in the Dyconit Overlord component. These policies enable the system to adjust its behaviour based on the current workload and performance metrics, ensuring optimal performance and consistency. Section 4.6 describes how the Dyconit system deals with faults. In Section 4.7, the prototype generic Dyconit consistency model is classified within the application-oriented perspective, specifically under the dynamic/optimistic subcategory. Finally, Section 4.8 discusses the design process and alternatives.

4.1 Generic Dyconit Model Requirements

This section defines the core Functional Requirements (FRs) and Non-Functional Requirements (NFRs) for the Dyconit system based on the findings of Chapter 3. This activity falls in stage (1) of the AtLarge Design Process (41).

Our objective is to develop a highly adaptable system capable of accommodating diverse

communication patterns within an EDA. This entails efficiently handling the addition and removal of nodes, ensuring seamless integration of varied data sources, and facilitating dynamic adjustments to workflow configurations. Additionally, the system should offer users a range of consistency options tailored to their specific needs, allowing them to strike a balance between data accuracy and system performance based on their operational requirements. To achieve this overarching goal, we leverage the existing requirements of the Conit model and the Dyconit model, making necessary modifications to ensure generality. Specifically, we extend requirements that are bound to Dyconits working for MVEs and introduce new requirements that cater to the distinctive communication patterns prevalent in EDAs.

Previously, the Dyconit system was predominantly designed for client/server systems, where clients make requests to servers and receive responses. Dyconits are employed in this context to maintain consistency between distributed environments housing multiple copies of the data. However, EDAs feature a distinct architecture wherein components communicate through events and commands that trigger actions in other components. These systems exhibit a diverse range of architectures and communication patterns, resulting in varying consistency requirements depending on the specific use case at hand. For instance, some EDAs prioritize the order of events, while others suffice with eventual consistency. Therefore, the same set of requirements cannot be applied to EDAs without modification to account for the differences in architecture and consistency requirements.

Building upon our understanding of the unique challenges posed by EDAs and the need for modified requirements, we now transition to defining the FRs enabling the expansion of the Dyconit system to work effectively within the domain of EDAs.

4.1.1 Functional Requirements

FR1 Event-driven architecture support: The system should accommodate different communication patterns, beyond the client/server architecture of the original Dyconit system.

In Chapter 3, we have identified that EDAs can be built using different communication patterns. The dyconit system should be able to accommodate each pattern to support various types of applications in various domains.

FR2 Range of consistency bounds: The system should support a range of consistency bounds to allow users to choose the level of consistency that best suits their needs.

Different applications and use cases may have different consistency requirements, depending on how critical it is to have the most up-to-date and accurate data. Consistency bounds are a way of measuring how well a system can handle different levels of consistency between different data sources or replicas. Consistency bounds are defined by two parameters: staleness and numerical error. Staleness measures how frequently the data sources are updated and how much delay there is between them. Numerical error measures how much the data values differ between the data sources.

FR3 Configurable initial bounds: Software engineers should be able to set their own initial consistency bounds. This feature would allow users to set consistency bounds based on their specific application requirements.

The system can accommodate different needs and allow users to choose the initial level of consistency that best suits their requirements by providing a range of consistency bounds. For instance, an e-commerce website might need fast updates and accurate values for transactions, so it would initialise the consistency bounds with low staleness and low numerical error values. On the other hand, a chat application might tolerate some delay and variation in chats, so it would initialise the consistency bounds with high staleness and high numerical error values. Hence, the system should be flexible enough to support both types of use cases.

FR4 Dynamic policies based on application-centric consistency requirements: The system should adjust its policies based on the current event throughput to ensure optimal performance.

This requirement is important for the system to be able to handle changing workloads and ensure optimal performance. By dynamically adjusting its policies, the system can respond to changes in activity and maintain the desired level of consistency. For instance, if there is a sudden increase in activity, the system can prioritize availability for all users, even if that means allowing for some level of inconsistency.

FR5 Real-time interactive system support The system should handle nodes joining or leaving the topology while it is running.

In EDAs, nodes should be able to join or leave the topology at any time, and the system should be able to handle these changes. For example, when a new node joins the topology, it should notify the others and receive the current state.

FR6 Run-time monitoring The system should provide insight into how the dyconits are behaving at run-time, including metrics for consistency, performance, and other relevant indicators.

The ability to monitor the system during operation is important to ensure optimal performance and detect potential problems. By providing insight into the behaviour of the dyconits, the system can be adjusted to improve its efficiency and effectiveness. For example, by observing trends in consistency metrics, operators can determine if certain dyconits are becoming bottlenecks or if data inconsistency issues arise. Meanwhile, monitoring performance indicators can help identify areas where the system can be improved to increase throughput or reduce response times, ensuring that users experience the highest quality of service.

4.1.2 Non-Functional Requirements

NFR1 Maintain low latency in sending messages related to Dyconits
The purpose of this NFR is to reduce the delay in sending messages that contain information about the dyconits' state and boundaries. These messages are essential for achieving good performance gains from inconsistency. As such, the system must be designed to ensure that the maximum latency for the transmission of dyconit-related messages does not exceed a predefined threshold of 1000 milliseconds.

NFR2 Ensure low overhead for Dyconit-related communication
The goal of this NFR is to reduce the overhead in resource utilization and bandwidth consumption for dyconit-related communication, which can affect the system's performance and scalability. It is important to keep the number of messages related to dyconits low. Therefore, the system should be designed to limit the number of dyconit-related messages to no more than one per node every five seconds.

NFR3 Ensure stable performance through high throughput and low Latency
The goal of this NFR is to maintain stable system performance by ensuring high throughput and low latency. The system must be designed to dynamically adapt to changing workloads while consistently maintaining a minimum throughput of 1.5 requests per second.

4.1.3 Requirements Development

We follow an iterative process for requirement analysis (41), which means that our requirements change as we learn more about the problem. We give some examples of how we modified key requirements over time and why we did so.

At first, we set Non-Functional Requirements for the applications that will use the dyconit system. These are different types of requirements, as shown in the use cases in Section **??**. However, this was not a good way to define requirements for the dyconit system itself because they did not capture its main functionality and performance. Therefore, we decided to focus the functional and non-functional requirements for the dyconit system on its own features.

We changed Requirement FR2 (Range of consistency bounds) to have only two consistency dimensions instead of three, as in the original conit implementation. We realized during the design phase that Apache Kafka already ensured order. To avoid having a useless consistency dimension, we removed it from our design.

We added Requirement FR6 (Run-time monitoring) after creating the first design of the generic Dyconit system. We realized that the Dyconit Overlord (Section 4.3) already needed to monitor the behaviour of the generic dyconits system. Therefore, we could easily extend this to allow users some feedback on how the system is performing. This is also useful for evaluating the system.

4.2 High-Level Overview of the Generic Dyconit System

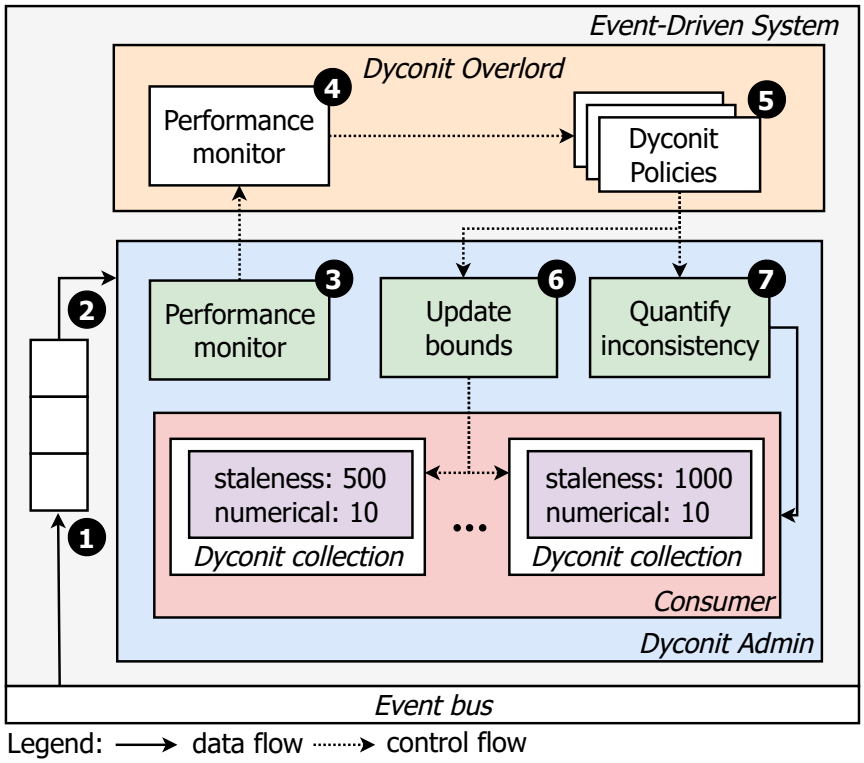


Figure 4.1: The high-level overview of the Dyconit system, which consists of two main components: the Dyconit Overlord and the Dyconit Admin. The Dyconit system optimistically bounds inconsistency in EDAs by allowing for different consistency requirements for different nodes. The Dyconit Overlord monitors the system performance, maintains the global collections of dyconits, and dynamically adjusts the policies for each dyconit. The Dyconit Admin acts as a middleware between the event bus and the nodes, and quantifies and bounds the inconsistency in staleness and numerical dimensions. The figure also shows how the components communicate with each other and with the event bus and the nodes.

This section presents the design of a generic Dyconit system for optimistically bounded inconsistency in EDAs. Focusing on EDAs, which facilitate the development of loosely coupled and scalable systems through event-based communication and processing (**FR1**), the Dyconit system limits resource consumption, even when disseminating events to a large number of subscribers, by bounding inconsistency between each event and its corresponding subscribers at a fine-grained level.

Figure 4.1 depicts our design. The design comprises two primary components: the Dyconit Overlord and the Dyconit Admin. The Dyconit Overlord assumes the responsibility

of maintaining an overview of system performance (**FR6**), conducting accounting for node participation in different conits, and dynamically adjusting policies (**FR4**). The Dyconit Admin system acts as a middleware, specifically designed as a component between the existing event bus of the EDAs and the producer/consumer nodes. It requires no modification in the core functionality of these nodes, making it a seamless integration within the architecture.

Having presented the high-level overview of the Dyconit system, we proceed to examine its components in greater depth. Consumers pull events from the event bus according to their subscriptions to specific events (❶ in Figure 4.1). The Dyconit admin processes the events (❷) and periodically sends performance metrics like event and synchronisation throughput (❸). The Dyconit Overlord collects and processes the performance metrics from all Dyconit Admins (❹). The policies describing the behaviour of the dyconits are adjusted dynamically by the Dyconit Overlord using performance metrics (❺). The system updates inconsistency bounds dynamically, based on the policies stored at the Dyconit Overlord (❻). To bound inconsistency, the Dyconit Admin quantifies the inconsistency caused by each incoming event (❼), forwards it to the corresponding dyconit, and evaluates the consistency requirements for each node (**FR3**). If a node's bounds are exceeded, the Dyconit admin forwards all event updates to the other nodes in the *Dyconit collection*. The Dyconit collection consists of an arbitrary group of consumer nodes that all share the same work.

4.3 Components of the Dyconit System

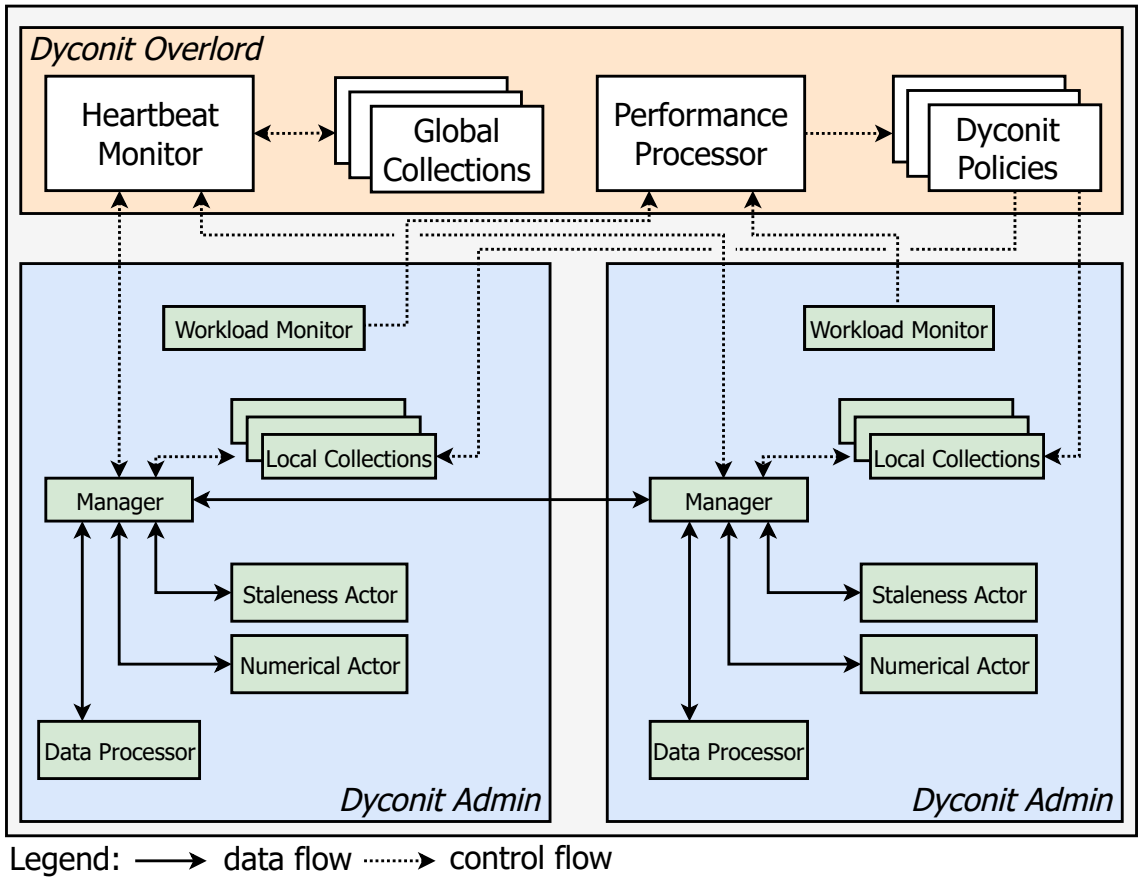


Figure 4.2: The internal structure and interaction of the two main components of the Dyconit system: the Dyconit Overlord and the Dyconit Admin. The Dyconit Overlord is responsible for monitoring the heartbeat of the nodes, storing the global collections of dyconits, processing the performance metrics, and applying the dyconit policies. The Dyconit Admin is responsible for managing the local collections of dyconits, bounding the inconsistency in staleness and numerical dimensions, and processing the data from other nodes. The figure also shows how the components communicate with each other and with other nodes in the EDA.

In this section, we zoom in on the two main components of the Dyconit system: the Dyconit Overlord and the Dyconit Admin. Figure 4.2 shows the internal structure and interaction of these components.

4.3.1 Dyconit Overlord

The Dyconit Overlord comprises four subcomponents: the Heartbeat Monitor, the Global Collections, the Performance Processor, and the Dyconit Policies.

The Heartbeat Monitor is responsible for sending periodic heartbeat messages to every node in the EDA. It obtains the information about the Dyconit Admins from the Global Collections. If the Heartbeat Monitor does not receive a response from a node, it removes the reference to this node from every collection that contains dyconits of this node. If this node becomes available again, the information is restored in the Global Collections and heartbeat messaging to this node resumes.

The Global Collections in the Overlord store information from each dyconit collection. A dyconit collection is a group of nodes that share the same dyconit. Hence, when a node joins the EDA, it also communicates with the Overlord which dyconit collections it is part of.

The Performance Processor gathers metrics on the state of the EDA. It also receives metrics from the nodes. Based on the available metrics, the Performance Processor decides whether to dynamically adjust the policies.

The dyconit policies apply to one or more dyconit collection in the Global collection. The policies are stored at the Overlord. Depending on the system throughput and the dynamic bound policy that is used, the consistency bounds of the dyconits are adjusted, and this update is communicated to all relevant nodes.

4.3.2 Dyconit Admin

The Dyconit Admin consists of six subcomponents: the Manager, the Workload Monitor, the Local Collections, the Staleness Component, the Numerical Component, and the Data Processor.

The dyconit manager has a dual responsibility. First, it is responsible for sending a heartbeat reply back to the Dyconit Overlord. Second, it checks for inconsistency in two inconsistency dimensions, based on the configured dyconits. If synchronisation is required for one of the consistency dimensions, the event is forwarded to the corresponding component. The Manager synchronises with other nodes by exchanging updates with the corresponding components on remote nodes. The figure shows only one remote node, but there can be arbitrarily many remote nodes to synchronise with.

The Workload Monitor is responsible for keeping track of the local performance of the node and analysing the performance and utilization of computing resources within. It sends this information to the Dyconit Overlord, which in turn can dynamically adjust the bounds based on the set policy.

The Local Collections store only the collections that are active on that node. They also have information about which other Dyconit Admins are in the same collection. This way, the Dyconit Admin can contact a remote node without going through the Overlord. If a node in the Local Collection is down, or a new node joins the collection, the Manager will update the Local Collection accordingly. If the bounds change, this change will reach the Local Collections.

The Staleness and Numerical Components are responsible for bounding optimistically bounding inconsistency. In case a bound is exceeded, it communicates the need to synchronise with all other nodes in their respective local collection through the dyconit manager.

The Data Processor is responsible for processing received events from other dyconit nodes.

4.4 Design of Consistency Bounding

The Dyconit system design meets **FR2** and **FR3** by using dyconits to limit inconsistency. Our design follows the work of Donkervliet et al. (9), which only considers two dimensions of inconsistency: staleness and numerical error. We do not use the third dimension of order error, which was used by Yu et al. (10) in their Conits, because event streaming platforms like Apache Kafka, which we use in our implementation, already ensure order (35).

In the Dyconit design for EDAs, software engineers can set the limits for staleness and numerical error for each event. This allows the application-centric perspective, where we have fine-grained control over the consistency of our application. A range of consistency options is offered by giving the limits a value between $[0, \infty)$. For a value of 0, the Dyconit system guarantees sequential consistency and for a value greater than zero, we use a form of eventual consistency. In the Dyconit design, each event produced has a specification which dyconit is affected, and a weight associated with it to enable numerical error limiting.

To bound staleness, each node keeps track of the last time writes were pulled from each of its neighbours in the local collection. Let t_i be the timestamp of the i-th event, and let T_i^j be the last time node j pulled writes from node i. Let S be the staleness bound, such that any event must have a view of the data that is at most S time units behind the most recent write. Then the staleness error of node r before processing the event can be calculated as:

$$E_r = \max_{i=1}^{n}(t_i - T_i^r) \tag{4.1}$$

This formula gives the maximum difference between the timestamp of any event and the last time node r pulled events from that node. If $E_r \leq S$, then node r can continue without violating the staleness bound. Otherwise, node r must pull writes from all nodes in the local collection before continuing processing.

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. Let w_i be the global weight of the i-th write, and let U_i^j be the set of locally unseen events by node j after performing the i-th event. Then the numerical error of node j after the i-th event can be calculated as:

$$E_i^j = \sum_{k \in U_i^j} w_k \tag{4.2}$$

This formula gives the numerical error after a given event. If $E_i^j \leq N$, then node j can continue without violating the staleness bound. Otherwise, node j must pull writes from all nodes in the local collection before continuing processing.

To bound inconsistency, we use the same reactive approach as Donkervliet uses in the original Dyconits (42). This means that we check for staleness and numerical order one after another. This reactive approach has the drawback of causing more synchronisation latency, but in return, we make the system more predictable and therefore easier to study. For each event, we trigger the staleness limiting mechanism first and then trigger the numerical error limiting mechanism. We synchronise with other nodes that are part of our local collection as soon as a consistency limit is exceeded. The implementation details are described in Section 5.1.3.

4.5 Dynamic Policies

The Dyconit Overlord design meets the requirements of **FR4** and **FR6** by employing a performance processor that adjusts the dyconits residing in the Global Collections according to the chosen policy. The performance processor periodically queries the Global Collections' nodes for their event production/consumption and internode synchronisation throughput. The local workload monitors report this information to the performance processor. The dyconit affects the consistency and communication overhead among the nodes. Decreasing the bounds improves the consistency, but also increases the number of messages required for synchronisation. Conversely, increasing the bounds reduces the number of messages, but also lowers the consistency. The performance processor can change the bounds based on the number of events produced, using the dynamic bound policy. The dynamic policy determines how the dyconits are adapted to achieve a balance between consistency and communication overhead. While software-engineers can create their own policies, we present three examples of policies and discuss them in the sections below.

4.5.1 Simple Policy

This policy adjusts its consistency bounds based on both event consumption throughput and synchronisation throughput. The consistency bounds are increased or decreased by a fixed percentage, depending on whether the throughput exceeds or falls below a certain threshold. By increasing the consistency bounds when the throughput is high, the policy allows for more flexibility and scalability. By decreasing the consistency bounds when the throughput is low, the policy ensures more accuracy and reliability.

4.5.2 Moving Average Policy

This policy adapts its consistency bounds based on the event consumption throughput's moving average over a window of 3. It compares the average with a threshold and adjusts the bounds accordingly. If the average is above the threshold, the bounds are increased to reduce the overhead of synchronisation. If the average is below the threshold, the bounds are decreased to at most their original values. This policy is more suitable for scenarios with bursty event consumption than the simple policy, as it can dynamically respond to changes in throughput. However, it does not change the bounds if the throughput is consistently low.

4.5.3 Exponential Smoothing

This policy uses exponential smoothing on event consumption throughput to adjust its consistency bounds. It computes a weighted average of the current and previous values, using a smoothing factor (α). The higher the smoothed value, the larger the bounds, which reduces the overhead of synchronisation. The lower the smoothed value, the smaller the bounds, which improves the accuracy of consistency. This policy is able to swiftly adapt to fluctuations in throughput and lessen the impact of outliers, due to its sensitivity to recent input changes and weighted averaging.

4.6 Real-time Interactive System Support

The design of the Dyconits system complies with **FR5** through the inclusion of two functionalities. These functionalities encompass the dynamic topology adaptation mechanism and the heartbeat monitor. The dynamic topology adaptation mechanism enables the system to automatically modify its network topology in response to environmental changes. This mechanism guarantees that new nodes can join the topology and existing nodes can exit without causing communication deadlock. For instance, it prevents situations where a node is waiting for a response from a node that is no longer present in the topology. The heartbeat monitor assumes the responsibility of continuously monitoring the health and availability of system components. It accomplishes this by regularly emitting heartbeat signals and verifying timely responses from other components. This functionality aids in the detection and resolution of issues or failures within the system.

4.6.1 Dynamic Topology Adaption Mechanism

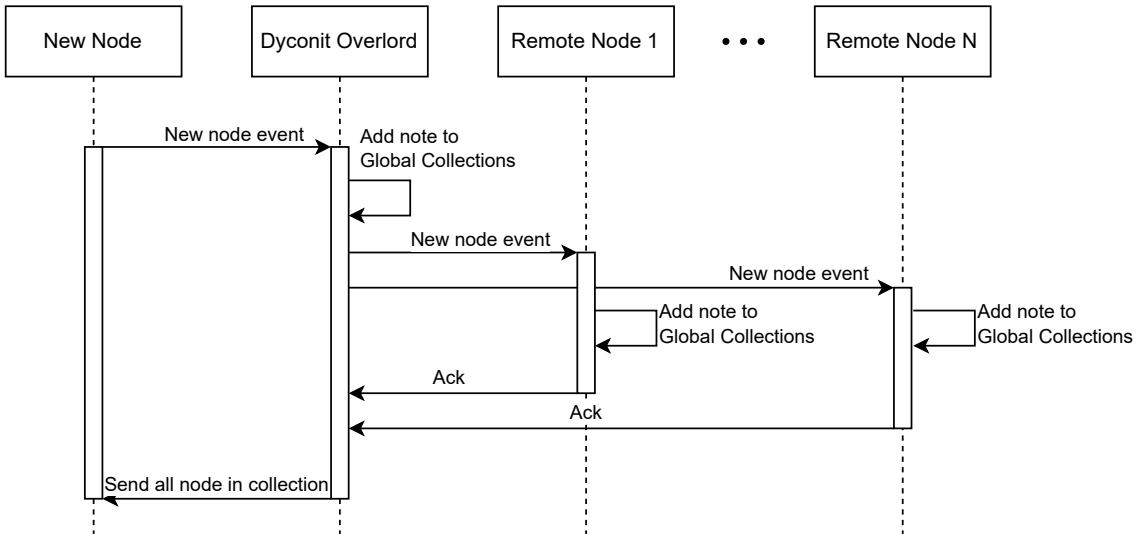


Figure 4.3: Sequence diagram of the dynamic topology adaptation mechanism for Dyconits. The diagram shows how a new node joins the topology and communicates with the Dyconit Overlord and the existing remote nodes.

As shown in Figure 4.3, the dynamic topology adaption mechanism consists of a sequence of steps that enable a new node to join or leave the topology and communicate with the Dyconit Overlord and the existing remote nodes. However, this scenario assumes ideal communication conditions without failures or delays. In reality, we employ various

techniques such as retry mechanisms, timeouts, and duplicate detection to cope with these challenges, as we discuss in detail in Chapter 5.

4.6.2 Heartbeat Monitor

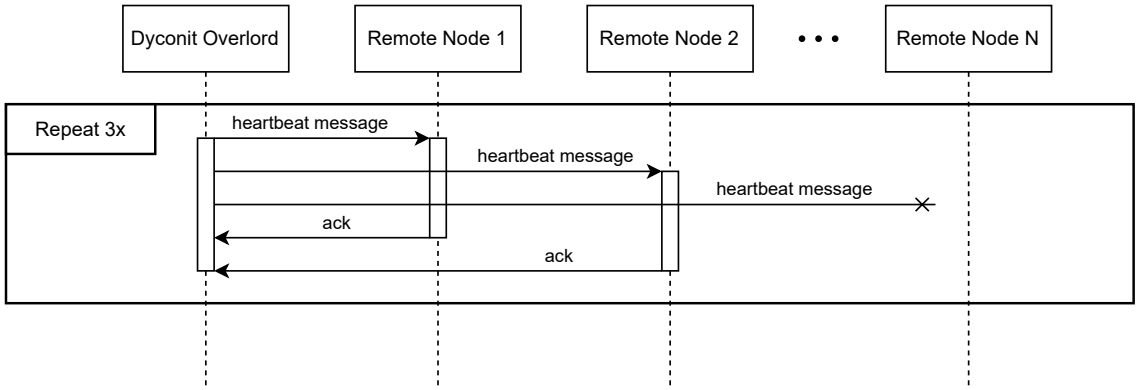


Figure 4.4: Sequence diagram of the heartbeat mechanism to monitor the status of nodes in the global collections.

The designed heartbeat mechanism periodically sends a request to each node in the global collections residing in the Dyconit Overlord to check if they are still alive. Figure 4.4 illustrates a simple sequence diagram of this mechanism. If a node fails to respond to three consecutive requests, the Dyconit Overlord assumes that it is down and notifies all other nodes via the sequence shown in Figure 4.3.

4.7 Consistency Model Classification

The first aspect to determine is the placement of the prototype generic Dyconit consistency model within the various perspectives outlined in Chapter 2. Dyconits itself belongs to the application-oriented perspective, as it enables tailoring consistency levels to specific subsets of operations, rather than adopting a one-size-fits-all approach. Moreover, the original Dyconits falls under the dynamic/optimistic subcategory within this perspective. This subcategory allows for temporal inconsistency, thereby reducing synchronisation overhead and improving performance. Additionally, it facilitates the adjustment of set bounds during execution based on a predefined policy.

Comparing the classification of the original Dyconits with the desired characteristics of the generic Dyconit model, we observe that we fulfil the requirements necessary for the

dynamic/optimistic subcategory within the application-centric perspective. For instance, the generic Dyconit model can control consistency at the application level (Section 4.3), specifies boundaries that permit a certain degree of inconsistency (Section 4.4), and incorporates a mechanism for dynamically adjusting these boundaries according to predefined policies (Section 4.5).

4.8 Design Process and Alternatives

We reached this design through an iterative design process following the principles of the AtLarge design vision (41) Our process involved repeated ideation and analysis of numerous designs, assessing their level of innovation as new solutions to the problem and their pragmatism in terms of feasibility for implementation in the specific conditions of use.

In this research, we have designed a system that uses Dyconits to enable optimistic inconsistency in general event-driven systems. Our design focuses on how consumers communicate with each other to cope with the challenges of latency and availability. However, this design choice restricts our system to scenarios where consumers are homogeneous and can share workloads among themselves. A more general system that also considers the roles of the event broker and the producer would introduce more complexity and challenges, which are beyond the scope of this research. Therefore, we suggest some directions for future work to extend our system. One possible idea is to leverage the information available at the broker level. For example, in Kafka, the API provides information about how far behind the consumer is from the producer. This information can be used to adjust the rate of event production and consumption dynamically, depending on the consumer's load and availability. This way, the system can balance the trade-off between consistency and performance by adapting to the changing conditions of the event-driven system.

5

Integrating the Generic Dyconit System in to Event-Driven Systems

In this chapter, we address the third research question (RQ3). We translate the requirements from the previous chapter into an implementation and provide context on the choices made during implementation. To validate the feasibility and effectiveness of our proposed Dyconit system for EDAs, we introduce Hestia. Drawing inspiration from Greek mythology, Hestia is named after the Greek goddess of the hearth, home, and domesticity, who represents the centrepiece of a harmonious household and family in myth. Similarly, our system serves as the central force that maintains consistency among various components in EDAs. This analogy highlights the significance of stable, reliable consistency mechanisms in the world of EDAs.

We implement a functional prototype that embodies the core aspects of the design[1]. We first outline the implementation decisions and their relation to the design requirements in Section 5.2. We describe the technical specifications and architecture of our prototype in Section 5.1. Finally, we discuss the challenges that we faced during the integration process in Section 5.3.

5.1 The Implementation of Hestia

This section presents a working prototype of a Hestia, a generic Dyconit system for EDAs, following the FRs from Chapter 4. First, we show how we support various communication patterns in EDAs (§5.1.1). Next, we explain how the Hestia Admin and Hestia Overlord

[1] `https://github.com/JurreBrandsen1709/dyconits_kafka`

manage their respective Dyconit collections (§5.1.2). Then, we discuss how we ensure Dyconit consistency in EDAs (§5.1.3). Finally, we describe how we enable the creation and dynamic adjustment of custom policies for consistency bounds (§5.1.5).

5.1.1 Event-Driven Architecture Support

As discussed in Section 3.4, Apache Kafka enables various communication patterns among applications. We use the `Confluent.Kafka` library for .NET, which offers a rich set of APIs and abstractions to interact with Apache Kafka. This library allows us to create and configure producers and consumers, as well as applications that can act as both.

Despite the availability of this rich API, the existing API was not sufficient to integrate Dyconits into our Kafka-based system. Dyconits introduces specific requirements and enhancements that had to be incorporated into the existing Kafka infrastructure. We augmented the functionalities of the `Confluent.Kafka` library to accommodate the unique features of Dyconits and ensure seamless integration with our system. The following modifications were made:

5.1.1.1 HestiaMessage

Kafka allows microservices to communicate through streams of events. An *event* in Kafka typically is a record of something that happened, such as a user action, a sensor reading or an entry log. Each event in the `Confluent.Kafka` library is called a `Message` and consists of a key and a value. The *key* is a unique identifier that determines which partition the event belongs to. The *value* is the actual data that the event carries, such as a JSON object or a plain text string. Determining the numeric value of an event requires a way to assign a weight to each event. For this reason, we extended the `Message` class of `Confluent.Kafka` to the `HestiaMessage` class. Now, users can also specify a weight for each event sent, in addition to a key and a value.

5.1.1.2 HestiaConsumerBuilder

Kafka uses the `ConsumerBuilder` method to create a consumer class. To ensure a new consumer communicates its consistency bounds to the `HestiaOverlord`, we created a wrapper, `HestiaConsumerBuilder`. This wrapper not only initializes the consumer but also carries the consistency bounds. It requires the `HestiaAdmin` port and consistency bounds as additional parameters.

On initialization, the class sends a `newAdminEvent` to the `HestiaOverlord`, containing the consistency bounds and admin port reference. The `HestiaOverlord` then informs all relevant `HestiaAdmins` (Section 4.6).

5.1.2 Keeping Track of Dyconit Collections

In Hestia, dyconits are managed by a central node called the `HestiaOverlord`. This node coordinates the communication and consistency levels among different components of the system. It also maintains multiple Global Collections, which define the policies and rules for each dyconit. The `HestiaOverlord` has several subcomponents called `HestiaAdmins`, which monitor and control individual consumers. They ensure that the consumers follow their specified consistency levels and report their dyconit states to the `HestiaOverlord`. They can also request adjustments to the consistency levels when needed. The main tasks of the `HestiaOverlord` are policy parsing, heartbeat messaging, active node tracking and admin client communication. When the `HestiaOverlord` is started, it initialises the `GlobalCollection` and then runs three concurrent methods: `ParsePolicies`, `ParseEvent`, and `ManageAdmins`. We will now explore each one of them in more detail.

5.1.2.1 Global Collection

The global collection uses the JSON data format, which is versatile and easy to integrate. The collection has a clear and intuitive naming convention, where the key matches the collection name and the values are nested key-value pairs. The collection has four consistent keys: `PolicyThresholds`, `PolicyRules`, `Hosts`, and `Bounds`. These keys have the following meanings:

- **PolicyThresholds**: These are the conditions or limits that trigger an action when met or exceeded.
- **PolicyRules**: These are the actions or outcomes that occur when a policy threshold is met or exceeded.
- **Hosts**: These are the hosts that belong to the same dyconit collection and share the same policies and rules.
- **Bounds**: These are the bounds for staleness and numerical error for each host in the collection, which define the acceptable range of data quality and accuracy.

5.1.2.2 ParsePolicies

The `HestiaOverlord` has to process the policies that define the dyconit collections, thresholds, and rules. These policies are stored as JSON files in a specific directory. The system first checks if the directory exists. Then, it reads each policy file and parses the JSON content. It extracts the important details about each dyconit collection and validates them for consistency and accuracy. It also updates the `Global Collection` with the new or modified information about the dyconit collections.

5.1.2.3 ParseEvent

The `ParseEvent` method in the `HestiaOverlord` class handles the messages received from the admin clients in the system. The messages contain various types of information or requests that require different actions from the `HestiaOverlord`. The method takes a message as a parameter, which can be a string or a serialized object. It first determines the message type, which can be one of several predefined types. Depending on the message type, it performs different actions:

- **newAdminEvent**: A new event has occurred in an admin client, such as joining or leaving a dyconit collection. The method extracts the event type and the associated data from the message and updates the state of the system accordingly. It may also trigger specific behaviours or notify other components based on the event type.
- **heartbeatResponse**: An admin client has responded to a heartbeat message sent by the `HestiaOverlord`. The method updates the heartbeat status of the corresponding node in the system, marking it as active and responsive.
- **throughput**: An admin client has sent its current throughput value to the `HestiaOverlord`. The method extracts the throughput value from the message and uses it to update the relevant data or trigger specific actions related to throughput management.

5.1.2.4 ManageAdmins

The `ManageAdmins` method in the `HestiaOverlord` class implements this functionality as shown in Algorithm 1. The method is an asynchronous task that runs continuously in a loop with a certain time interval.

Algorithm 1: Heartbeat Management

Result: Keep track of active nodes, remove inactive nodes (unless heartbeat message is received), and send heartbeat messages to admin clients

```
while true do
    Wait for a specific time interval;
    Check the heartbeat status of all nodes in each dyconit collection;
    foreach node do
        if heartbeat message is received from the node then
            Continue to the next node without removing it;
        end
        if node is inactive then
            Remove the node and notify other nodes;
        end
    end
    Send heartbeat messages to all admin clients in each dyconit collection;
end
```

The method iterates over each dyconit collection in the Global Collection and checks the heartbeat status of all nodes in each collection. For each node, it verifies if a heartbeat message has been received from it. If so, it skips the node and proceeds to the next one. If not, it checks if the node is inactive and removes it from the collection. It also notifies other nodes about the removal. After checking all nodes, the method sends heartbeat messages to all admin clients in each collection, requesting a response.

5.1.3 Enforcing Dyconit Consistency

Hestia relies on a core functionality that ensures dyconit consistency. This functionality is implemented by the `HestiaAdmin` class, which extends and utilises the `Confluent.Kafka` administrative client. The `HestiaAdmin` class employs several methods to enforce consistency. These include two methods for bounding Staleness and Numerical Error, as well as other methods that handle sending and receiving `syncRequests`. Furthermore, the `HestiaAdmin` also ensures that incoming events and local events are integrated correctly and decides if the consumer can commit its local events. The subsequent subsections elaborate on each functionality individually

Algorithm 2: Bound Staleness

Result: Return the local data after checking and updating the staleness bound for each port in the collection

```
result ← UpdateLocalData(localData, collectionName);
portsStalenessExceeded ← [];
foreach port in ports do
    timeDifference ← consumedTime - lastTimeSincePull;
    if timeDifference > staleness then
        portsStalenessExceeded.Add(port);
    end
end
if no ports exceed staleness then
    return result;
end
if optimisticMode then
    RequestUncommitedEventsAsync(portsStalenessExceeded, collectionName);
    return result;
end
else
    RequestUncommitedEvents(portsStalenessExceeded, collectionName);
end
return result ← UpdateLocalData(localData, collectionName);
```

5.1.3.1 Bounding Staleness

Algorithm 2 describes the BoundStaleness algorithm, which ensures that each node in a dyconit collection has events that are within a certain freshness range and avoids stale or outdated data. The algorithm requires four inputs: `consumedTime`, `localData`, `collection`, and `collectionName`. The `consumedTime` represents the time when the consumer processed the event. The `localData` consists of a list of local and uncommitted `ConsumeResults`. The `collection` is a JSON data structure that stores information about the local dyconit collection, such as the staleness bound, the ports of other nodes, and the last time since pull for each node. The `collectionName` identifies the name of the local dyconit collection.

The algorithm aims to bound the staleness error among consumers within a local dyconit collection. The algorithm involves the following steps:

1. **Get latest data from other consumers**
   We first update the local data with the latest data from the other consumers for the given dyconit collection. This ensures that we have a consistent view of the data before checking for staleness.

2. **Check for staleness bound violations**
   The second step is to compare the consumption time of the new event with the last interaction time with each other consumer in the local dyconit collection. The last interaction time is the time when the consumer last exchanged events with another consumer in the same collection. If the difference between the consumption time and the last interaction time exceeds the staleness bound, then there is a violation.

3. **Request uncommitted events**
   In the third step, the requester asks each consumer that has exceeded the staleness bound to send all their uncommitted events since the last interaction. After sending their uncommitted events to the requester, the consumers can commit them. Depending on the policy, the requester may operate in an optimistic mode, where it does not wait for the synchronisation process to finish.

5.1.3.2 Bounding Numerical Error

Algorithm 3 describes the BoundNumericalError algorithm, which ensures that each node in a dyconit collection has events that are within a certain numerical range to avoid too much inconsistency. A new event is an event that has an associated weight, which represents the amount of numerical error that it introduces. Our numerical error bounding mechanism is activated after the staleness bounding mechanism.

The algorithm aims to bound the numerical error among consumers within a local dyconit collection. The algorithm involves the following steps:

1. **Get latest data from other consumers**
   We first update the local data with the latest data from the other consumers for the given dyconit collection. This ensures that we have a consistent view of the data before checking for staleness.

2. **Check for numerical error bound violations**
   The second step is to compare the weight of the new event with the numerical error

Algorithm 3: Bound Numerical Error

Result: Return the local data after checking and updating the staleness bound for each port in the collection

```
result ← UpdateLocalData(localData, collectionName);
error ← calculateNumericalOrderError(collection);
if error > numericalBound then
    RequestUncommitedEvents(portsInCollection, collectionName);
end
return result ← UpdateLocalData(localData, collectionName);
```

bound for each other consumer in the local dyconit collection. The numerical error bound is a predefined threshold that determines the maximum allowable numerical error for each consumer. If the weight of the new event exceeds the numerical error bound, then there is a violation.

3. **Request uncommitted events**
   In the third step, the requester asks each consumer that has exceeded the numerical error bound to send all their uncommitted events since the last interaction. After sending their uncommitted events to the requester, the consumers can commit them.

5.1.4 Communication between HestiaAdmins

The `HestiaAdmin` that detects a bound violation must obtain all uncommitted events from the consumers in the same collection by sending a synchronisation request. These synchronisation requests are sent and received asynchronously, so they may arrive at different times. To synchronise them properly, we use Algorithm 4. This algorithm checks that the events received are not older than the events already processed by the consumer. If they are not, it merges the uncommitted events by taking the union of the local and received events. If the size of the union is different from the size of the local events, meaning that more events were received than are consumed locally, it sets the 'changed' boolean to true. This informs the consumer that synchronisation has taken place and that all processed and received events up to that point can be committed.

The `HestiaAdmin` keeps all relevant properties associated with a consumed event, such as the topic, partition, offset, and timestamp, by synchronising the entire `ConsumeResult` object rather than just the message content because we need the complete `ConsumeResult` to commit an event. The synchronisation process is facilitated by various wrapper classes,

Algorithm 4: Merge events

Result: The synchronised event collection and a boolean indicating whether any changes were made

```
if ReceivedEvents is not empty then
    filter ReceivedEvents to keep only items with the same topic as collectionName;
end
if ReceivedEvents.count < LocalEvents.count then
    changed ← false || _synced;
    return SyncResult(LocalEvents, changed);
end
MergedEvents ← LocalEvents ∪ ReceivedEvent;
changed ← MergedEvents == LocalEvents;
return SyncResult(MergedEvents, changed);
```

such as `ConsumeResultWrapper`, `MessageWrapper`, and `HeaderWrapper`. These wrappers allow synchronisation between `HestiaAdmin` objects by encapsulating the consumed messages and their associated metadata. Moreover, the SyncResult class encapsulates the outcome of synchronisation operations, providing a structured representation of the changes made during the process.

To compare local `ConsumeResults` and received `ConsumeResults`, we implemented a custom comparer class, `ConsumeResultComparer`, which enables comparison of consumed messages based on their offset and topic.

Furthermore, the decision to commit messages is based on the presence of uncommitted consumed messages and the result of the synchronisation process. The consumer class invokes the `CommitStoredMessages` method if there are uncommitted messages or any changes detected during synchronisation. This step ensures that the processed messages are permanently stored and will not be reprocessed in subsequent iterations or system restarts, contributing to data reliability and consistency.

5.1.5 Custom Policies for Adjusting Consistency Bounds Dynamically

One of the main challenges of implementing a system that supports dynamic consistency bounds is to provide a mechanism for software engineers to specify the conditions and actions that trigger the adjustment of the bounds according to the functional requirements of

the application. To address this challenge, we designed a policy-based approach that allows software engineers to create policies in JSON format that define the rules for changing the consistency bounds. A policy consists of several elements:

- The policy name, which indicates the goal and the scope of the policy.
- The collections that are affected by the policy.
- The thresholds that define the limits or targets for relevant metrics or variables. These can be used to measure the performance or quality of the application. For example, a policy may set a threshold for the consumer's throughput, which is the rate of processing messages from a topic. Other examples of thresholds are latency, availability, error rate, etc.
- The parameters that are needed for advanced policies. These can be used to fine-tune the policy behaviour or logic. For example, a policy may require a smoothing factor or a window size as parameters.
- The rules that specify the logic for adjusting the consistency bounds. Each rule has a condition and an action. The condition is a logical expression that evaluates to true or false based on the values of the metrics or variables. The condition format is '{variable} {operator} {threshold}' For example, a condition may be 'throughput $\leq$ 1000', which means that the throughput is below the threshold. The action is a function that calculates a new value for the consistency bound based on the operator and the value in the rule. For example, an action may be '+0.1', which means that the consistency bound should be increased by 0.1 units. The action applies to either the staleness or the numerical order error bound, depending on the collection type.

Listing 4 shows an example of a policy that applies to two collections and has one rule that increases the staleness bound by 0.2 if the throughput is lower than 500.
By using this policy-based approach, software engineers can express their functional requirements in terms of consistency bounds and their dependencies on various metrics or variables. The system then interprets and executes these policies at runtime to adjust the consistency bounds dynamically.

```
{
  "Policy Name": "Increase Staleness Bound",
  "Collections": [ "Collection_1", "Collection_2" ],
  "Thresholds": { "Throughput": 500 },
  "Rules": [
    {
      "Condition": "Throughput < Threshold",
      "Actions": [
        {
          "Target": "Staleness",
          "Operator": "+",
          "Value": 0.2
        }
      ]
    }
  ]
}
```

Listing 1: Sample JSON Policy

5.1.5.1 Calculating Throughput

One of the metrics that can be used to define policies for adjusting consistency bounds is throughput, which measures the rate at which a consumer processes messages from a topic. To calculate throughput, we present Algorithm 5, that outlines the process of computing the throughput per consumer for a given topic.

The process begins by retrieving the metadata associated with the given topic, which includes information about its partitions. For each partition, we proceed to extract the previous offset and timestamp from a dictionary that maintains a record of offsets for each partition. It is important to note that if no previous offset is found, we initialize the offset to 0. Subsequently, we calculate the current offset and timestamp for the partition in question. To ascertain the consumption rate, we compute the difference between the current and previous offsets and divide it by the time difference in seconds, thereby representing the rate at which the consumer processes messages from that specific topic partition. The consumption rate is then added to the cumulative value of the topic's throughput. By using this algorithm, we can measure the throughput of each consumer for each topic and use

this metric as an input for defining policies for adjusting consistency bounds dynamically.

5.2 Implementation Choices

This section explains the implementation choices, how they correspond to the design principles and how they fulfil the requirements specified in Chapter 4.

5.2.1 Selecting the Message Broker

We chose to use Apache Kafka as the message broker in our implementation of the Event-Driven Architecture. The selection of Apache Kafka stems from a comprehensive evaluation of various message broker solutions based on several criteria. Firstly, Apache Kafka can be easily integrated with C# applications using the `Confluent.Kafka` library [1], which is a .NET client for Apache Kafka that supports .NET Core and .NET Framework. The `Confluent.Kafka` library provides a producer and a consumer class that can be used to send and receive messages from Kafka topics. Secondly, Apache Kafka provides robust metrics and monitoring capabilities. Apache Kafka ensures that the messages it processes are arranged in a strict sequence. This feature of Apache Kafka makes it appropriate for our implementation, as it eliminates the order consistency error. Thirdly, the distributed nature of Apache Kafka enables horizontal scalability, accommodating high message throughput and ensuring efficient handling of peak loads. Lastly, the decision to utilize Apache Kafka over Azure Event Hub was driven by the desire to avoid vendor lock-in, ensuring the flexibility to migrate our system to different infrastructure providers in the future if necessary. Unlike Azure Event Hub, which is a Platform as a Service (PaaS) offering that requires us to rely on Azure's managed infrastructure and services, Apache Kafka is an open-source software that can be deployed on any cloud or on-premise environment. This gives us more control over the configuration and customization of our message broker solution, as well as lower operational costs and higher portability.

5.2.2 Selecting the Programming Language

.NET was our choice of language, primarily for its profound support for event-driven programming. Its event model simplifies event-related tasks, positioning it favourably against Java or Scala. Secondly, C# in the .NET ecosystem offers an expressive syntax, fostering rapid development and reducing the learning curve in comparison to other languages.

[1] `https://github.com/confluentinc/confluent-kafka-dotnet`

Thirdly, .NET is widely used in EDA applications that Info Support produces, demonstrating its relevance and effectiveness for our specific needs. Finally, the ability to integrate effortlessly with our chosen message broker, Apache Kafka, further validated this choice.

5.2.3 Selecting the Policy format

For crafting custom dynamic policies, JSON was our choice of data format. The decision hinged on several factors: the readability of JSON, its intuitive key-value structure, and its ubiquitous adoption among software engineers. This choice was bolstered by the need for a format that was both human-friendly and machine-optimized, streamlining the policy creation process.

5.3 Implementation Challenges

In this section, we discuss the main challenges that we faced while implementing Hestia. We categorize these challenges into three high-level groups: integration with Kafka-based System, data processing and validation, and asynchrony. We also describe the specific challenges that we encountered within each group and how we addressed them.

5.3.1 Integration with Kafka-based System

We used the `Confluent.Kafka` library as our message broker for implementing Hestia. However, this library did not support some of the features that we needed, such as staleness and numerical order error bounds. To address this issue, we extended and modified some of the existing classes and methods of the library, such as the `Message` class and the `ConsumerBuilder`. We also added our own classes, `HestiaAdmin` and `HestiaOverlord` that managed the communication with the Kafka infrastructure and the dyconits and their consistency bounds.

5.3.2 Data Parsing, Validation, and Calculation

Hestia receives various types of messages from the admin clients, such as policy messages, heartbeat messages, throughput messages, etc. Each message type required a different action from the system, such as updating the Global Collections, triggering specific behaviours, or notifying other components. Therefore, we had to implement a robust message parsing and handling mechanism that correctly interpreted and responded to different message types. We used JSON parsing techniques and exception handling mechanisms to address this challenge.

5.3.3 Asynchronous Methods

Hestia is designed to run continuously in the background, performing certain tasks at specific time intervals, such as calculating throughput, sending heartbeat messages, or adjusting consistency bounds. These tasks were asynchronous, meaning that they did not block the main thread, and they could run in parallel. However, this also introduced some challenges regarding synchronisation and ordering of tasks. The main challenge that we faced was scheduling and executing periodic tasks: Hestia had to perform various tasks at regular intervals without blocking the main thread or interfering with each other.

Algorithm 5: Calculate Throughput

Result: The calculated throughput per consumer

if *consumer is null* **then**
  return
end
Initialize offsets as an empty dictionary;
Retrieve the partitions for the specified topic from the metadata;
foreach *partition in partitions* **do**
  Create a topicPartition object for the current partition;
  Retrieve the previous offset for the topic partition;
  if *previousOffset is unset* **then**
    Set previousOffset to 0;
  end
  Add the topicPartition and its corresponding previousOffset to the offsets dictionary;
end
Delay the execution for 5 seconds;
Initialize topicThroughput as 0.0;
foreach *partition in partitions* **do**
  Create a topicPartition object for the current partition;
  Retrieve the previous offset and timestamp from the offsets dictionary;
  Retrieve the current offset for the topic partition;
  if *currentOffset is unset* **then**
    Set currentOffset to 0;
  end
  Set currentTimestamp to the current time;
  Calculate the consumption rate as the difference between currentOffset and previousOffset divided by the time difference in seconds;
  Add the consumption rate to topicThroughput;
end
Create a JSON object containing the calculated topic throughput, admin port, and topic information;
Send the JSON object as a message over TCP;

6

Evaluation of the Performance of a Generic Dyconit System: Experimental Design and Results

Hestia is a prototype implementation of the Dyconit consistency model for event-driven systems, which allows for fine-grained and dynamic control of consistency levels in event-driven systems. In this chapter, we present the experimental design and evaluation of Hestia, answering RQ4. In Section 6.1, we present how we designed synthetic workloads based on interviews with domain experts. We describe the experiment setup in Section 6.2. We explain the experiment deployment in Section 6.4. We discuss the experiment configuration in Section 6.5, including how we set the boundaries, variables, and delays for the experiment. Section 6.6 outlines the research questions, hypotheses, and metrics covering topics from topology impact to sensitivity analysis. Lastly, Section 6.7 presents the findings and their implications.

6.1 Designing Synthetic Workloads Based on Interviews with Domain Experts

In this section, we present how we designed synthetic workloads to evaluate Hestia in realistic settings. These workloads reflect the real-world challenges and scenarios that domain experts face in event-driven environments. We obtained these insights from semi-structured e-mail interviews with five domain experts who work with event-driven systems in different fields, such as e-commerce, transport, and finance (Appendix 8.1). Based on their responses, we derived three representative workloads that capture different aspects

of how event-driven systems behave. This approach was more feasible and reliable than integrating the prototype into live applications, which would pose practical challenges such as potential disruptions, complex integration requirements, and dependencies on external factors. Similarly, obtaining real-world event traces was difficult due to privacy concerns, data availability limitations, and the proprietary nature of such traces. By leveraging the expertise and experiences of these domain experts, we aim to enhance the authenticity and applicability of our experimental evaluations, as well as to simulate and evaluate the behaviour of Hestia in a controlled environment.

From the email interviews with the domain experts, we identified several key factors that influence event-driven systems. We observed that all workloads shared similar patterns: they experienced periods of high and steady event rates, regardless of the sector. We also discovered that the priority of events varied depending on the type of workload. For instance, logging events were low-priority but very frequent. This was relevant for our experiment because our Dyconit implementation can offer different levels of consistency for different events. Therefore, we could apply more strict consistency for high-priority events than for low-priority logging events. The interviews also revealed variations in the sizes of events processed by these systems. Logging events are small, ranging from 10 to 50KB. Higher priority events tend to be larger in size. One interviewee said that they had to cut a priority event into smaller pieces because it exceeded the Kafka max message size, while in other domains this was not necessary. Another interviewee said that their priority events were around 1MB in size. These events are clearly larger than the logging events. The final insight we can draw from these interviews is the distribution between logging and priority events. This tended to be an 80/20 ratio, where 80% of the events were logging events, and the remaining 20% were priority events.

Based on the insights from the interviews, we designed three distinct workload patterns to evaluate the performance of our prototype under various scenarios. These patterns represent the diverse types of workloads that the domain experts discussed in their interviews. In each workload, events are distributed across two different topics, with an 80/20 ratio, to emulate the varying priorities of events. Several interviewees mention this is standard practice. The topic with 80% of the events processes smaller events with a size of 1 KB, while the topic with 20% of the events processes larger events with a size of 10 KB. We chose these sizes to have a clear difference between priority and normal logging events. The smaller events are more frequent and less complex, while the larger events are rarer

and more detailed. This reflects the trade-off between timeliness and completeness that the domain experts expressed in their interviews. Using events of these sizes allows us to avoid splitting them into multiple events, which would introduce additional overhead and complexity in our prototype.

Each workload runs for a duration of 300 seconds. For simplicity and clarity in evaluation, we assigned a weight of 1.0 to each event. Although our prototype supports any non-zero positive weight, we chose a uniform weight of 1.0 to ensure a clear and straightforward analysis of the system performance under different workloads, avoiding potential complexities and ambiguities arising from differing event weights. We can classify the different synthetic workloads we used into three categories:

W1 The first workload is a constant rate where the Kafka consumers are undersaturated. This workload is meant to represent the average load on the system during non-peak moments in the interviews. This pattern simulates a normal and predictable traffic situation, where the producers and consumers are balanced, and the system is stable.

W2 The second workload is a fluctuating workload, alternating between undersaturation and over saturation of the Kafka consumers. The transition between the minimum and maximum is linear. This pattern simulates a moderate and gradual variation of traffic over time, aligning with observations of distinct patterns and peaks throughout the day, as shared by some experts in their systems. The fluctuation repeats multiple times to simulate different stages of high and low throughput of events. This will result in a saw-tooth like pattern, which can test adaptability of our Dyconit prototype of the system under varying load conditions.

W3 The third workload is a workload where the Kafka consumers are constantly oversaturated. This workload is meant to simulate the worst-case scenario of the system, where the demand exceeds the supply and the system cannot keep up with the incoming events. This pattern simulates a sudden and extreme spike of traffic, which could be caused by an unexpected increase in events. This will result in a large backlog of events in the Kafka topics, which can affect the performance and consistency of the system.

6.2 Experiment setup

Having presented how we designed the synthetic workloads based on interviews with EDA experts at Info Support, we present the remainder of the experimental setup. Here, we explain what network topologies were used in the experiments to confirm that Dyconits are generically applicable in various scenarios. Additionally, we discuss which policies were tested to evaluate the impact of dynamic consistency on performance.

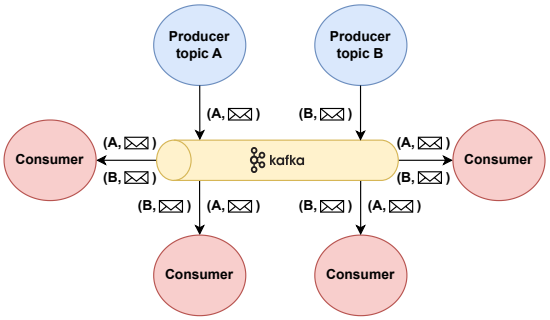
6.2.1 Network Topologies' Impact on Performance and Behaviour

The topology of the event-driven system is another important factor to consider. It describes how events are generated, delivered, and processed by various components in the system. The topology affects the performance and consistency of the system, as it influences aspects such as network delay, bandwidth usage, and fault resilience. To evaluate the prototype Dyconit consistency model, we need to measure its performance under different topologies that reflect the characteristics of general event-driven systems. By conducting experiments with various topologies, we aim to verify whether the prototype can support multiple event-driven systems as claimed. This analysis will enable us to understand the prototype's adaptability and generalizability, ensuring that it can cope with different topologies that are typical in real-world event-driven settings.

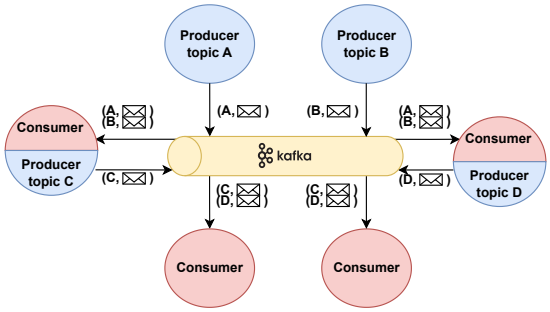
In our experiments, we considered a scenario where all consumers perform the same task and can exchange data with each other. If a service experiences a lag, it can request the processed data from another service via the broker. Then, the service can resume its consumption from the broker with the offset that corresponds to the received data, instead of continuing from the previous offset. This way, no extra processing time is needed, as we assume that the data has already been processed by other services.

For our experiments, we selected two topologies that we identified as the most common ones in the analysis of event-driven systems and their requirements in Chapter 3. These are the star-topology and the multi-hop topology. The star-topology consists of a central broker that acts as a hub for all events in the system. Producers send events to the broker, which then distributes them to consumers based on their subscriptions. The multi-hop topology consists of a network that forms a chain or a tree for event routing. Producers send events to one or more brokers, which then forward them to other brokers or consumers based on their subscriptions. By comparing the prototype Dyconit consistency

Figure 6.1: Two common ways of routing events in event-driven systems: the star and the multi-hop topologies. In the star topology, a central broker connects all producers and consumers. In the multi-hop topology, a network of brokers and consumers forms a chain or a tree. The figure also shows the topics and services involved in each topology



(a) Star topology with two producers and four consumers.



(b) Multi-hop topology with four producers and four consumers, two of which are co-located in a single service.

model under these two topologies, we aim to evaluate the generality of the Dyconit model in event-driven systems

Figure 6.1 illustrates the star and multi-hop topologies, respectively. The star topology consists of two producers and four consumers, while the multi-hop topology comprises four producers and four consumers, two of which are co-located in a single service. This service consumes events from topics A and B and produces events for topics C or D, depending on the input. A single broker mediates the event flow between all producers and consumers, as indicated by the arrows.

6.2.2 Effects of Dynamic Policies on Consistency

The final aspect to examine is the policy that determines how the prototype Dyconit system adapts its boundaries during execution. The policy defines when, how often, and by how much the boundaries should be changed based on either the synchronisation throughput or the message throughput. The right policy could have a significant impact on the trade-off between consistency and performance achieved by Hestia, as it determines how responsive and adaptive it is to changing conditions.

To explore how different policies affect Hestia, we experimented with four types of policies: none, simple, moving average, and exponential smoothing. These policies are defined in Section 4.5 and their JSON representations are given in Appendix **??**. Our main interest in using different policies is to see how they affect the effectiveness of Hestia in achieving

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

its goals of providing tailored consistency levels and reducing synchronisation overhead. By varying the policies, we aim to observe how the boundaries of the Dyconits change over time and how that influences the consistency and performance metrics of the system. We also aim to compare and contrast the advantages and disadvantages of each policy, as well as to identify the best policy or combination of policies for different scenarios and requirements.

6.3 Metrics and Data Collection

We used logging during the execution to collect data and measurements for the experiments. The prototype logs data about the system itself, such as the number of messages consumed, the throughput of event messages and synchronisation messages, the current consistency bounds and the latency between sending and receiving a synchronisation message. We analyse this data to understand how the system is performing and to check if it meets the NFRs we defined in Section 4.1.2.

In Table 6.1, the definitions and units of the five metrics used to evaluate the performance of our system in our experiments are shown: consumer lag (CL), message throughput (MT) and overhead throughput (OT). The first metric, consumer lag (CL), indicates how consistent the system is at time t. It is defined as the maximum delay between any two services in the system, measured in seconds. A lower CL means that the services have a more up-to-date view of the system state. The second metric, message throughput (MT), compares how efficient the system is in handling events at time t. It is defined as the number of event-related messages sent or received by the system per second. A higher MT means that the system can process more events in a given time. The third metric, overhead throughput (OT), measures how much overhead is caused by the dyconit system at time t. It is defined as the number of synchronisation-related messages sent or received

by the system per second. A lower OT means that the system does not incur much extra communication cost when using Dyconits.

6.4 Experiment Deployment

A central feature of our experimental system deployment is that each service runs in its own dedicated container. This mirrors the approach taken in real-world applications, where services are typically isolated in individual containers, often orchestrated by systems like swarm or Kubernetes. The deployment of our system is orchestrated using Docker Compose[1], an essential tool for defining and managing containerized Docker applications. This section delves into the architecture, the reasons behind using Docker, and the hardware on which the experiments are run.

- **System Architecture:** Our system consists of five main components: Apache ZooKeeper, Apache Kafka Broker, Custom Workload, Dyconit Overlord, and Application services. Crucially, **each of these components is deployed in its own distinct container**. Below, we describe each component in more detail.
- **Apache ZooKeeper:** As a centralized service for maintaining configuration, naming, distributed synchronisation, and more, ZooKeeper operates within its dedicated container. It's deployed using the image *confluentinc/cp-zookeeper:7.3.0* and acts as the coordination layer for the Kafka broker.
- **Apache Kafka Broker:** Kafka, a distributed event streaming platform, is responsible for communication within the system. It, too, is contained in its own environment, deployed via the image *confluentinc/cp-kafka:7.3.0*, and works in tandem with ZooKeeper.
- **Workload service** Depending on the specific experiment executed, the workload service runs one of three workloads, all within its separate container.
- **Dyconit Overlord Service** Dedicated to dyconit-related messaging, the Dyconit overlord service processes messages, tracks application services, and adjusts bounds based on active policies – all within its container environment.
- **Application Services** Running in their individual containers, these services embody custom applications that contribute to the experimental setup. They're developed in

[1] `https://docs.docker.com/reference/`

.NET and originate from specific Dockerfiles, interacting with one another and the Kafka broker to create a variety of network topologies.

6.4.1 Motivation for using Docker and Docker Compose

Embracing Docker and Docker Compose for our experimental deployment ensures each service's autonomy in a dedicated container. Containers encapsulate applications along with their environments, isolating them from external factors, which in turn guarantees a consistent and controlled execution environment. Although the services are simulated on the same network — introducing minimal latency in communication — we've implemented a random processing delay for each event at the service level to simulate real-world conditions. Refer to Section 6.5 for more details. Ultimately, Docker and Docker Compose enable a controlled, reliable deployment, replicating the complexities of distributed systems with realistic delays.

6.4.2 Hardware and Software Configurations

The host system is a Dell Latitude 5531 laptop, which is a standard model provided by Info Support to all its employees. It has a 12th Gen Intel Core i7-12800H processor, running at 1.80 GHz, 32.0 GB of RAM and 1 TB of SSD storage. It runs Windows 11 Enterprise as its operating system. The host system is connected to a Wi-Fi network with a bandwidth of 200 Mbps.

In this project, we use several NuGet packages to manage our dependencies and enhance our functionality. *NuGet* is a .NET package manager that facilitates the straightforward installation and updating of libraries from a unified repository. The required packages are specified in the `.csproj` file - an XML document outlining the project's settings and references. We integrated the following packages: Confluent.Kafka (1.9.3), which serves as a client for Apache Kafka (a distributed streaming platform); Newtonsoft.Json (13.0.3), a .NET JSON framework; and Serilog (3.0.1), a .NET application logging library.

6.5 Experiment Configuration

Our prototype has many tunable variables that affect its performance. One of these variables is the bounds, which is a mechanism that ensures synchronisation between two services. The bounds can be either staleness or numerical error, which depend on the elapsed time or the difference in numerical values between the services. Another variable is the

policy, which defines the throughput thresholds and the actions to take when the current throughput is above or below them. The policy also determines how the bounds are adjusted. To obtain specific and reliable results from our experiments, we fixed all of these variables for each experiment. In addition, we introduced a random processing delay to the consumption of each event to simulate the real-world scenarios. This delay represents the time it takes for a service to process an event and update its state. This subsection explains our choices of variables in more detail.

6.5.1 Configuration of the Boundaries

In the experiments, we use different predefined bounds for different topics. For the topic that consumes the logging events, we set the staleness bound to 15000 ms and the numerical error bound to 25. These settings allow for lenient consistency promises, as missing or delaying a few logging messages is not critical, according to the expert interviews. However, for the topic that handles priority events, we use stricter bounds to ensure tighter consistency assurance. Therefore, the staleness bound is 10000 ms and the numerical error bound is 10. This configuration is beneficial because it offers a higher level of consistency guarantee, which is vital for priority events. This way, we can ensure that high-priority data is processed with the urgency it requires, reducing the risk of losing or delaying important information.

6.5.2 Configuration of the Variables in the Policies

We use policies to dynamically adjust the boundaries based on changing workloads or other environmental factors. Policies are designed to be highly customisable, as explained in Section 4.5. We have designed three policies for our experiments. To compare the effect of each policy, we kept them as similar as possible. The system evaluates the throughput every 5 seconds. We set the throughput threshold to a fixed value of 1.2 for all policies. Preliminary experiments had shown that this was an optimal value for the threshold, given the modelled event processing delay we introduce. Depending on the delay and the workload, this threshold could be exceeded or not. If the threshold was exceeded, we increased the numerical bound by 1, allowing for an additional message lag compared to the other services, and we increased the staleness bound by 1000 ms, allowing for a longer time lag than before. By choosing these actions, we maintained realistic boundaries. Other values tested in preliminary experiments resulted in either too loose or too tight boundaries, or required to synchronise every consumed message.

6.5.3 Modelling Event Processing and State Update with Random Delays

We make a clear distinction between priority events and normal events. Priority events are those that require immediate attention and processing, such as urgent orders or critical alerts. Normal events are those that can be processed at a lower priority, such as logging. The processing delay for priority events is between 300 and 900 ms for slow services and 200 to 600 ms for normal services, while for normal events this is between 30 and 90 ms for slow services and 20 to 60 ms for normal operating services. These delays are proportional to the importance of the events and reflect the different service-level agreements (SLAs) that may exist in real-world scenarios.

We simulate the processing of events and the updating of state by introducing random delays for the application services. We use two different configurations for the delays, based on the input from domain experts. Two out of the four services in both topologies will have delays ranging from 30(0) to 90(0) ms, representing services that take longer to update due to various reasons. The other two services will have delays ranging from 20(0) to 60(0) ms, representing faster services. These delays are always the same for every experiment, as we use a dictionary of predefined uniformly distributed values that depend on the message we consume. These delays serve several purposes. First, they add realism, as real-world systems are not instantaneous and have inherent processing and network communication delays. Second, they model the unpredictability common in distributed systems. Third, they stress-test the system and help us understand its behaviour under suboptimal conditions. The selected delay range thus helps us get a general understanding of the system's performance across various conditions.

6.6 Experiment Design

To organize our experiments and results, we divided them into three groups based on their objectives and focus:

G1 Performance Evaluation: This group includes the experiment that compares the performance of Hestia against a baseline system without optimistic inconsistency. The goal of this group is to demonstrate the benefits of adding optimistic inconsistency to a distributed system, specifically targeting NFRs for low latency in sending

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

dyconit related messages and maintaining high throughput and low latency of the system.

G2 System Configuration Evaluation: This group includes the experiments that demonstrate the generality of Hestia by applying different policies, workloads, and topologies. The goal of this group, closely related to the NFRs, is to show that the system can handle various scenarios and requirements with different configuration options, ensuring low overhead for the dyconit related messages and maintaining system performance stability.

G3 System Behaviour Evaluation: This group includes the experiments that investigate how the system behaves under different conditions, such as scalability and sensitivity. This group's NFRs aim to understand how the system adapts to changing workloads, parameter settings, and performance trade-offs.

The different experiments are carefully selected based on the NFRs proposed in the design on the Dyconit system in Section 4.1.2. As outlined above, our goal is to show that the prototype we developed maintains a low latency in sending dyconit related messages, ensures low overhead for the dyconit related messages, maintains a high throughput and low latency of the system, and ensures that the performance remains stable. Table 6.2 summarizes the configurations of each experiment in terms of topology, workload, policies, and metrics. We explain each experiment in detail in the following subsections.

We collect a range of metrics in these experiments. These metrics are relevant and important for evaluating the performance of distributed systems because they reflect how well the

Table 6.3: Variables and collected metrics for the Optimistic Inconsistency Impact experiment.

Experiment	Topology	Workload	Policies	Metrics
§6.6.1 Dyconit Impact	T1	W2	P1	CL, MT

systems can cope with the challenges of scalability, reliability, and efficiency in a dynamic and heterogeneous environment, thus adhering to our NFRs. The Message throughput is the number of messages delivered per second by the producer to the consumers. The Overhead throughput resembles the number of dyconit related messages are sent per second. Consistency lag is the difference in time between when a consumer receives an update and when all other consumers receive the same update.

6.6.1 Optimistic Inconsistency in Hestia

Our primary interest lies in understanding the implications of integrating the dyconit consistency model into an event-driven system. Specifically, we want to understand how this model affects the consistency and performance of a system, especially when duplicated consumers have the capability to share processed work among each other. The objective of this experiment is to examine the influence of the dyconit model, particularly the optimistic inconsistency mechanism, on Hestia's performance. We intend to quantify the trade-offs: How might system performance and data consistency interact and possibly counterbalance each other in our prototype?

To conduct a robust analysis, we utilise workload 2, which modulates between undersaturation and over saturation of consumers. This specific workload is chosen as it enables us to test Hestia under varying operational conditions, thus providing a holistic view of the system's behaviour. Given that our central concern is discerning differences brought about by the inclusion or exclusion of Dyconits, we opted for the simplest policy for comparison purposes.

For this experiment, we have two distinct system versions:

- **Version A:** A system with the optimistic inconsistency mechanism activated.
- **Version B:** A baseline system that functions without the optimistic inconsistency feature.

Table 6.4: Variables and collected metrics for the Topology Impact experiment.

Experiment	Topology	Workload	Policies	Metrics
§6.6.2 Topology Impact	T1,T2	W2	P1	CL, MT, OT

To ensure a fair comparison, we made certain that both these versions have analogous configurations, which encompass topology, workload, and message size specifics. Table 6.3 furnishes detailed information about the configurations and metrics adopted for this experiment.

6.6.2 Hestia's Generality Across Topologies

The focus of this experiment revolves around the Hestia prototype and its applicability across different system topologies. Distinct topologies possess varied attributes such as connectivity, diameter, routing complexity, and fault tolerance. These characteristics could potentially influence the performance and behaviour of distributed systems, including aspects related to consistency.

Our overarching aim is to emphasize the versatility and adaptability of the Hestia prototype, positioning it as a system capable of functioning across diverse network structures and communication patterns. To reinforce this claim, we have chosen to test our prototype in two representative topologies: star and multi-hop. These were selected due to their capacity to represent a wide spectrum of possible scenarios while being relatively easy to understand conceptually.

As highlighted in Table 6.4, we utilise workload 2, which modulates between undersaturation and over saturation of consumers. This specific workload is chosen as it enables us to test Hestia under varying operational conditions, thus providing a holistic view of the system's behaviour. Through this systematic evaluation across different topologies, our objective is to discern the efficacy of our prototype in adapting and excelling under varied network conditions and communication modes.

6.6.3 Hestia's Generality Under Varied Workloads

The primary goal of this experiment is to determine the impact of different workloads on the performance and optimistic inconsistency of the Hestia Prototype. Different workloads can have unique consequences on distributed systems, influencing aspects like congestion,

Table 6.5: Variables and collected metrics for the Workload Impact experiment.

Experiment	Topology	Workload	Policies	Metrics
§6.6.3 Workload Impact	T1	W1, W2, W3	P1	CL, MT, OT

latency, and load balancing.

We will test our prototype using three distinct workloads:

- **W1:** A standard workload characterized by a consistent message rate.
- **W2:** A bursty workload with a fluctuating message rate that alternates between high and low peaks.
- **W3:** A peak workload with a message rate that exceeds the system's capacity.

The specific parameters for each of these workloads are detailed in Table 6.5. Our choice of a star topology is driven by its simplicity and efficiency, ensuring direct connections among all services. The rolling average policy was selected because of its capacity to adjust to shifting workloads.

6.6.4 Hestia's Generality Under Varied Policies

Our primary goal is to explore the impact of different policies on the Hestia prototype. We are interested in determining how these policies may impact Hestia's adaptability, consistency, and performance. To this end, we have chosen to examine three specific policies in contrast to a scenario with no policy. The policies under consideration are:

- **No Policy:** As the name suggests, this approach maintains the bounds without any adjustment.
- **Simple Policy (P1):** This approach imitates the behaviour of the TCP protocol. The bounds increase by a fixed amount when there aren't any pending messages, and decrease by half when there's an overflow of pending messages.

Table 6.6: Variables and collected metrics for the policy Impact experiment.

Experiment	Topology	Workload	Policies	Metrics
§6.6.4 Policy Impact	T1	W2	none, P1, P2, P3	CL, MT, OT

Table 6.7: Variables and collected metrics for the Sensitivity Analysis experiment.

Experiment	Topology	Workload	Policies	Metrics
§6.6.5 Sensitivity Analysis	T1	W2	dynamic range	CL, MT, OT

- **Rolling Average (P2):** This policy sets a dynamic threshold based on a rolling average of message rates. Optimistic inconsistency is applied only when the volume of pending messages surpasses this threshold.
- **Exponential Smoothing (P3):** It leverages a weighted average between the present value and the preceding smoothed value to modify the bounds, granting more significance to recent data.

Additional variables related to our study can be found in Table 6.6. Our emphasis lies predominantly on contrasting the capability of these policies in managing irregular workloads, often witnessed in real-world setups. Such workloads introduce variable and unpredictable message rates, potentially leading to spikes in system load. As a representative sample, we've incorporated the peak workload W3 in our experiment, marked by a high message rate surpassing the system's threshold.

6.6.5 Policy Parameter Effects on Hestia's Performance and Consistency

Our goal is to examine how the system responds to different policy settings. We used two sets of parameters: one for the scales with thresholds $(0.8, 1.0, 1.2, 1.4)$ and another for the scaling with the increase in consistency bounds $(+1, +2, +4, +8)$. We selected these numbers based on preliminary tests that showed they provided a good contrast of policy objectives. The thresholds cover both normal and extreme cases, while the increments in bounds allow us to measure the system's flexibility to both minor and major changes. With this method, we created 16 variations of the moving average policy, each with a unique pair of parameters, enabling a thorough assessment.

Our goal is to help software engineers design policies that suit different situations. These situations often involve a trade-off between consistency and performance. We investigate how Hestia reacts to different parameter settings, which are the core of our research. These parameters, such as staleness bounds, numerical error bounds, and message weight thresholds, determine when the system should deliver a message optimistically.

Table 6.7 shows our experimental setup in detail. We want to understand how these parameter changes affect the system's performance and optimistic inconsistency. We keep other variables, such as topology, workload, and policy, constant, so that we can focus on the parameter variability. We choose a star topology for a clear network structure, a fluctuating workload for a realistic operational environment. This way, we can isolate the effects of parameter modifications and understand their implications on the system performance. The data from different parameter settings will reveal how they shape the system's performance and optimistic inconsistency.

6.7 Experiment Results

This section presents the experiment results of our prototype. We evaluated the performance of our prototype under different scenarios. We used several metrics to measure the effectiveness and efficiency of our prototype, such as accuracy, latency, throughput, and scalability. We also conducted scalability and sensitivity analysis to demonstrate the robustness and practicality of our prototype.

6.7.1 Performance Analysis of Optimistic Inconsistency in Hestia

MF1 By using the Dyconit consistency model, Hestia reduces the inconsistency by about 70% for both priority levels, effectively bounding inconsistency.

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

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

Figure 6.2 illustrates the impact of Dyconits on consumer lag for both priority and normal events across varying workloads. Consumer lag represents the time difference between event production and consumption by clients. In this analysis, we compare two configurations: one with Dyconits disabled and another with Dyconits enabled. Dyconits is a dynamic consistency adjustment system that tailors consistency levels to event priorities.

Starting with the evaluation of priority events, which demand robust consistency to ensure data accuracy and freshness, we observe distinct trends in the two configurations. In the

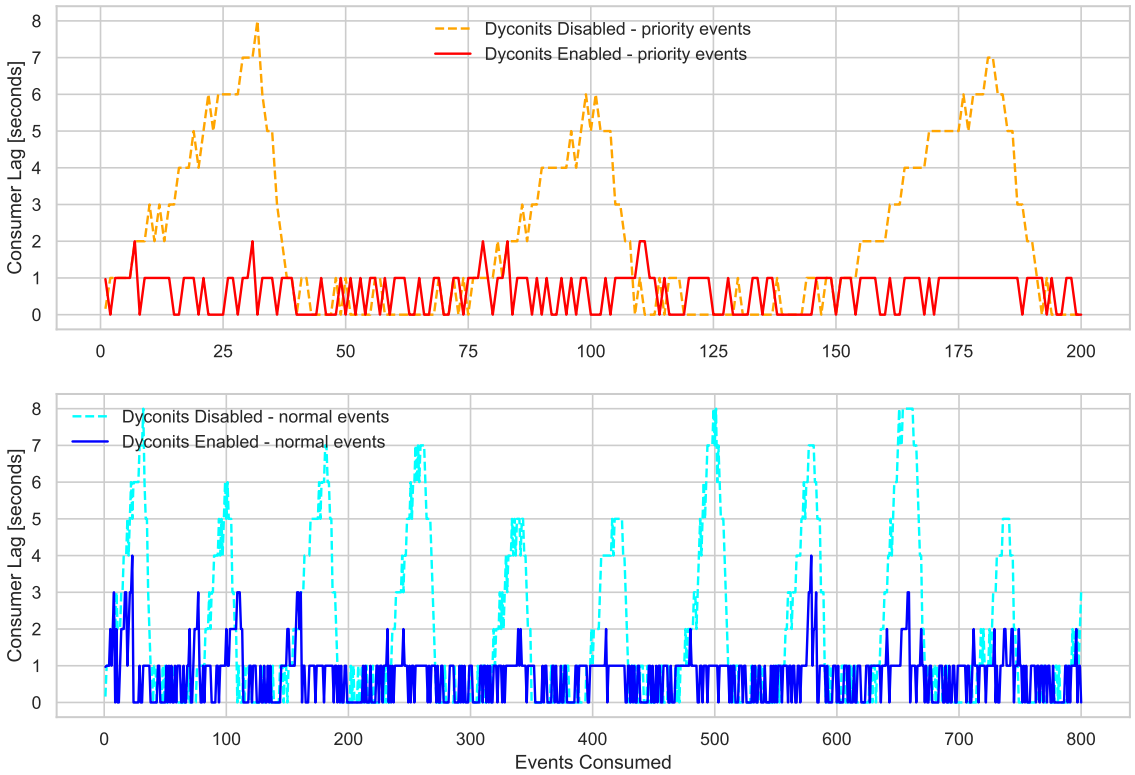


Figure 6.2: The effect of Dyconits on the maximum consumer lag for priority and normal events under a fluctuating workload.

setup without Dyconits, the maximum consumer lag reaches 7.304 seconds, with a mean of 2.1224 seconds. This configuration exhibits noticeable inconsistency, as evidenced by the escalation in consumer lag with increasing production rates of priority events. The maximum consumer lag in the Dyconits-enabled configuration, on the other hand, is only 2.273 seconds, around 68.9% lower than the maximum lag without Dyconits. The mean consumer lag in the Dyconits-enabled setup is 0.5101 seconds, which is approximately 76.0% lower than the mean lag in the absence of Dyconits. These figures demonstrate that Dyconits can substantially mitigate consumer lag for priority events, particularly during high saturation, resulting in faster and more accurate event delivery to clients.

Shifting our focus to normal events, which tolerate some inconsistency and staleness, we notice another noteworthy contrast between the configurations. In the Dyconits-disabled setup, the maximum consumer lag for normal events is 8.621 seconds, while the mean lag is 1.905 seconds. In contrast, the Dyconits-enabled configuration achieves a maximum lag of 3.913 seconds, a reduction of approximately 54.6% compared to the maximum lag

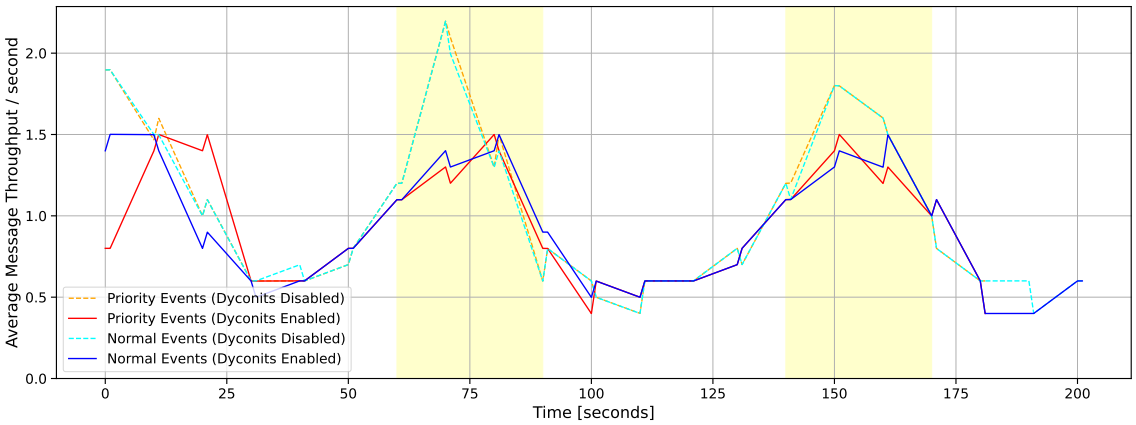


Figure 6.3: Comparison of average message throughput over time for priority and normal events, with and without Dyconits enabled.

without Dyconits. The mean consumer lag for normal events with Dyconits enabled is 0.6118 seconds, a difference of around 67.9% compared to the mean lag without Dyconits. These statistics underline Dyconits' effectiveness in dynamically adjusting consistency levels based on event priorities, thereby optimizing system performance and reducing synchronisation overhead.

Overall, Figure 6.2 demonstrates that Dyconits plays a pivotal role in restricting consumer lag for events of varying priorities under fluctuating workloads. By tailoring consistency levels to event importance and system conditions, Dyconits enhances event delivery efficiency and accuracy, contributing to an improved user experience and system performance.

The throughput trends over time for different event priorities and a fluctuating workload are shown in Figure 6.3, with a comparison between the scenarios with and without Dyconits enabled. Throughput is a key performance metric in this evaluation, as it measures the number of events processed per second. The figure shows that the average throughput that can be achieved is influenced by the number of produced events. Moreover, when the consumer reaches the saturation point, the differences between the system with and without dyconits are the largest, as indicated by the highlighted sections. This is likely because the system with dyconits has to synchronise more frequently at this point as it receives more messages, which in turn affect the numerical value and staleness.

The figure reveals a significant difference, with the highest throughput being about 46.26%

more in the scenario without Dyconits for priority events than in the scenario with Dyconits enabled. Likewise, the average throughput is roughly 12.06% more when Dyconits are disabled.

Moving on to the normal event scenario, here, the highest throughput in the scenario without Dyconits enabled is around 46.49% higher than in the scenario with Dyconits enabled. The average throughput is also about 10.80% higher when Dyconits are disabled.

The graph illustrates a consistent trend of higher mean throughputs for both priority and normal events in scenarios where Dyconits are disabled. It is clear that Dyconits incur extra overhead due to the synchronisation between consumers, leading to lower processing capabilities for normal events. This trade-off between performance and consistency suggests that while enabling Dyconits improves uniformity, it also reduces the overall system performance.

6.7.2 Performance Analysis of Hestia Across Star and multi-hop Topologies

MF4 Hestia is applicable to two common topologies that share the same basic building blocks as other topologies, such as tree, mesh, or ring. Thus, our results can be generalized to other scenarios that involve different communication patterns between services.

MF5 Hestia achieves a significant reduction of consumer lag in multi-hop topologies, lowering the average consumer lag of this configuration by a factor of 40.

Figure 6.4 depicts variations in maximum consumer delay across four configurations, highlighting topology and Dyconits activation effects. A comparison between star and multi-hop topologies is made. The impact of enabling or disabling Dyconits within these setups is also explored.

In the star topology, disabling Dyconits leads to a maximum delay of around 7.3 seconds (avg. 2.1 sec). Enabling Dyconits reduces maximum delay to 2.2 seconds (avg. 0.5 sec). This reveals a 68.98% increase in max delay and 76.0% increase in avg delay with Dyconits disabled. In the multi-hop topology without Dyconits, max delay surges to 77.1 seconds (avg. 41.2 sec). Enabling Dyconits cuts max delay to 4.4 seconds (avg. 0.9 sec), improving max delay by 1620.96% and avg delay by 4157.14%.

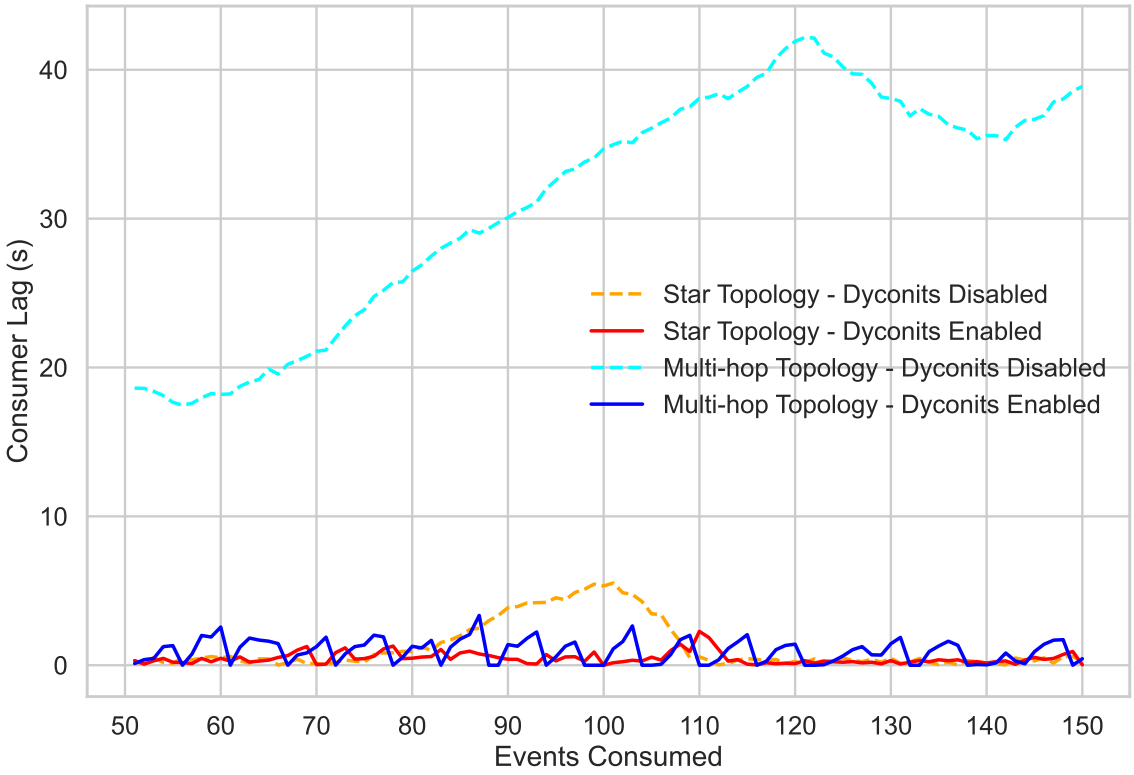


Figure 6.4: Maximum consumer lag for star and multi-hop topologies with and without Dyconits.

Notably, in the multi-hop setup without Dyconits, a substantial inconsistency arises. After 50 messages, this configuration lags about 20 seconds behind due to inter-service dependencies. Dyconits mitigates this by providing fresher data from faster services, ameliorating the problem. Comparing multi-hop with Dyconits to star with Dyconits, the former exhibits slightly more inconsistency, attributable to multi-hop's reliance on other services. This emphasizes the need for tailored bounds and policies for different topologies.

Figure 6.4 validates the effectiveness of our prototype for using Dyconits in event-driven systems. We selected the star and transitive topologies based on our earlier findings in Chapter 3 which showed that they are the basic building blocks for other topologies. Therefore, our results can be generalized to other scenarios that involve different communication patterns between services. However, we also acknowledge that there are challenges and limitations in applying our prototype to more complex topologies, such as routing, consistency, scalability, and fault tolerance issues. These are some of the open problems that we

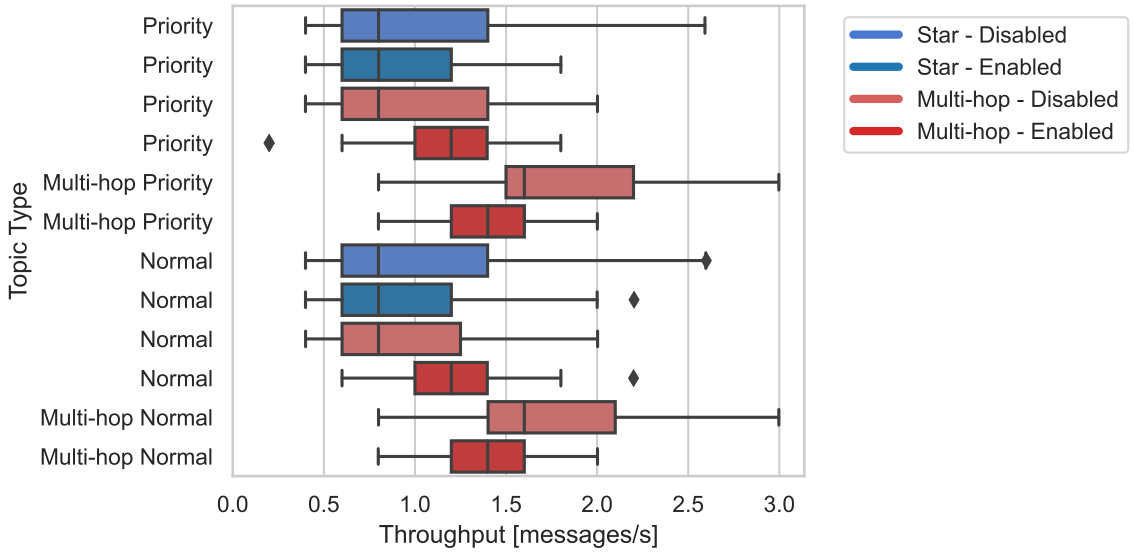


Figure 6.5: Average message throughput for star and multi-hop topologies with and without Dyconits.

plan to address in our future work.

Figure 6.5 shows the average throughput of messages per second for all configurations used in this experiment. We distinguish the messages sent from the consumer/producer service (intermediary) to the consumer in the multi-hop topology as the 'multi-hop' topics. These messages are different from the priority or normal events, and combining them would skew the results. The figure shows that our results from Section 6.7.1 do not apply to the multi-hop topology. Previously, we concluded that the throughput of messages is higher without Dyconits because of no overhead from synchronisation messages. However, this is not true for the multi-hop topology. The intermediary, which consumes and produces events, is a bottleneck for the messages that reach the end consumer via the multi-hop topics. The throughput here is low, while once the messages reach the end consumer, the throughput is much higher than with Dyconits. At the end consumer, the argument of the previous experiment applies. There is no communication afterwards, so the throughput can simply continue. For the multi-hop configuration with Dyconits, the intermediary is not the bottleneck. This also explains the drastic differences in results of Figure 6.4. By frequently synchronising with the other service that also acts as an intermediary, it does not lag and also prevents the other service from lagging afterwards.

For the star topology, we observe that disabling Dyconits leads to higher throughput than

enabling Dyconits, regardless of the priority level of the events. This is consistent with our previous findings in Section 6.7.1 that Dyconits introduce some overhead in processing synchronisation messages. Comparing the star topology with Dyconits disabled and priority enabled to those with Dyconits enabled and priority enabled, the former exhibits approximately 30% higher max throughput and around 12% higher mean throughput. Likewise, when comparing Dyconits disabled with normal priority to Dyconits enabled with normal priority, the former demonstrates roughly 23% higher max throughput and a slight 1.6% higher mean throughput.

For the multi-hop topology, the results are more nuanced. For priority events, enabling Dyconits resulted in about a 10% decrease in max throughput and a 14% increase in mean throughput. This suggests that Dyconits have a trade-off between achieving higher consistency and lower latency at the cost of some throughput reduction. For normal events, however, Dyconits led to around an 11% higher max throughput and almost 19% higher mean throughput. This indicates that Dyconits can improve both consistency and throughput for normal events, which are less time-sensitive and more tolerant of latency variations.

If we compare the multi-hop topology with the star topology, we see a significant difference, especially in the gain the system makes when it uses Dyconits. For the star topology, there is a trade-off between consistency and better throughput, but that does not seem to be the case for the multi-hop topology in terms of throughput. This means that the multi-hop topology can achieve both high consistency and high throughput by using Dyconits, while the star topology has to sacrifice one for the other. This suggests that the multi-hop topology is more suitable for applications that require both properties. However, the multi-hop topology also has some drawbacks, such as higher network latency and more complex service dependencies. Therefore, the choice of topology depends on the specific requirements and characteristics of the application.

The overhead distribution for the different configurations is shown in Figure 6.6. This figure excludes the configurations without Dyconits, as they do not rely on Dyconits for synchronisation. The star topology has a lower average overhead than the multi-hop topology, with a reduction of about 16.74% for priority events and 20.38% for normal events. However, the star topology also has a higher maximum overhead, with an increase of 38.46% for priority events and 16.67% for normal events compared to the multi-hop topology. These differences indicate the performance variation between the two topologies, and

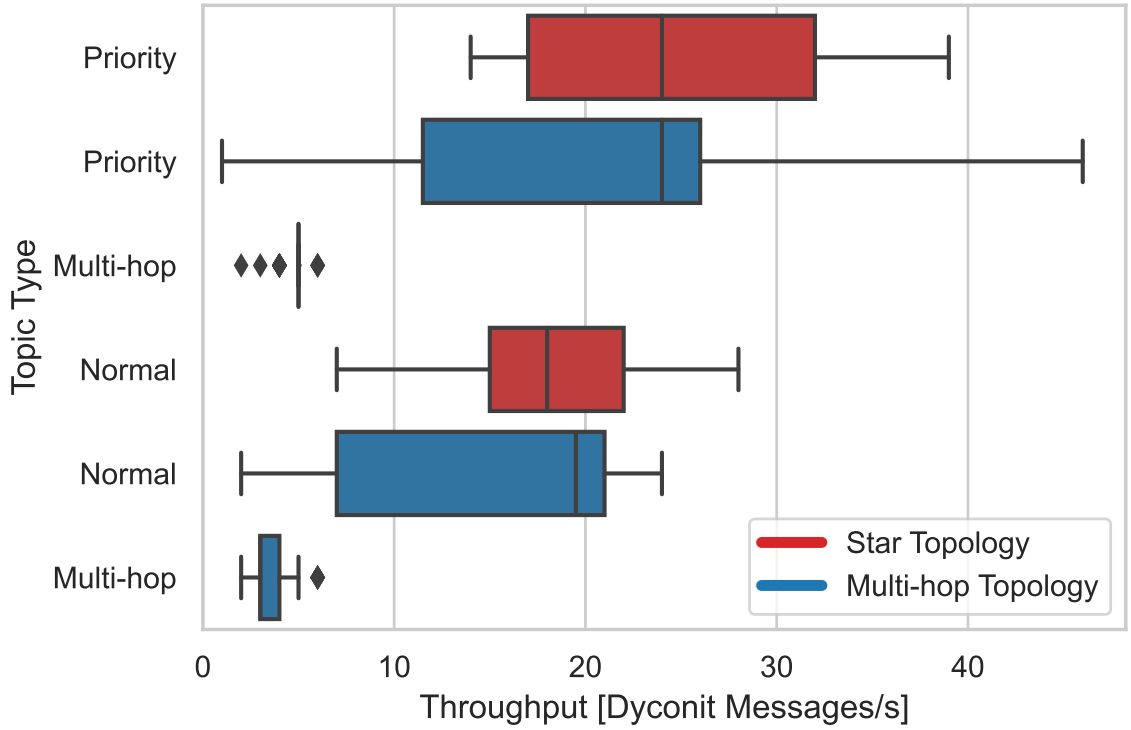


Figure 6.6: Average overhead for star and multi-hop topologies with Dyconits.

suggest that the star topology may offer some benefits in terms of overhead reduction.

One possible reason for this is that the star topology has fewer hops between the event source and destination, which reduces the latency and the bandwidth usage. Moreover, we observe that the overhead for the multi-hop topology has a wide range, especially for priority events. This could be related to the workload used for this configuration, which has a variable behaviour that may affect the priority bounds adjustment. We also confirm the same finding as in Experiment 6.7.1, where the priority events had a higher overhead, as they require stronger consistency guarantees.

Another interesting observation is the low overhead for the multi-hop events. This could be attributed to the fact that the intermediate service that is usually behind often synchronises, which makes the messages arrive at roughly the same time at all services, and reduces the need for synchronisation at the final consumer. However, this also means that the multi-hop events have a lower freshness than the priority events, as they may be delayed by the intermediate service. This trade-off between overhead and freshness is an important factor to consider when choosing a configuration for Dyconits.

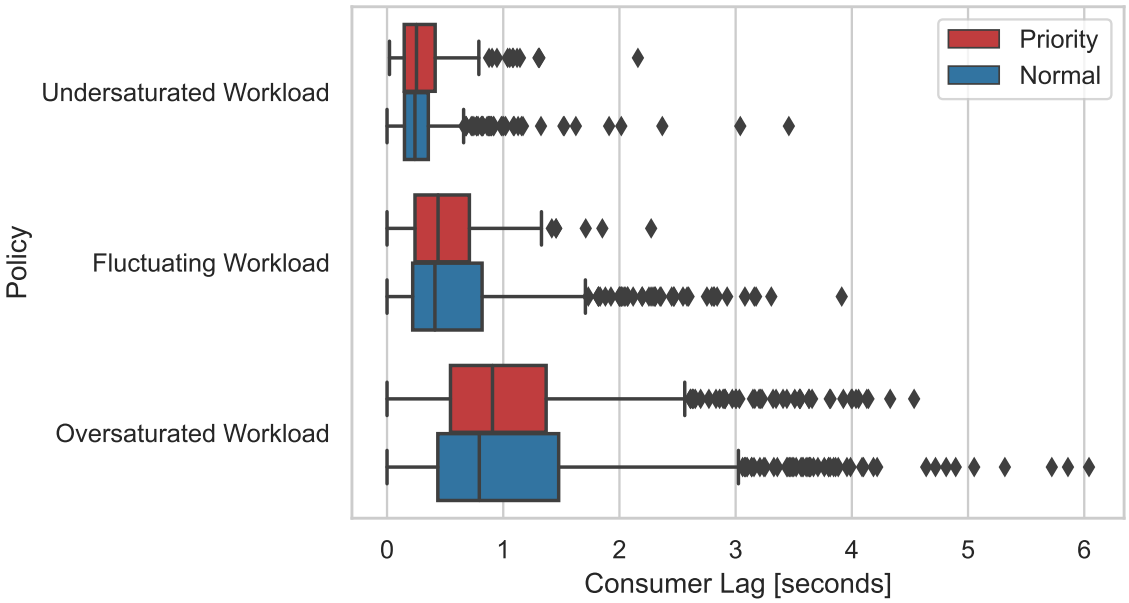


Figure 6.7: Distribution of consumer lag for priority and normal events under different workload situations.

6.7.3 Performance Analysis of Hestia Under Varied Workload

MF6 Hestia adapts to the workload saturation by balancing performance and consistency. It boosts performance within limits when the workload is oversaturated, and enhances consistency at the cost of performance when the workload is undersaturated. For instance, Hestia reduces the peak priority consumer lag by 25% compared to the normal events in an oversaturated load.

MF7 Hestia shows its flexibility and resilience by adjusting to different types of workloads, from simple to complex, and from homogeneous to heterogeneous. This illustrates the generality of our design and implementation.

Figure 6.7 shows the distribution of the consumer lag when handling priority and normal events in different workload situations. We observe that consumer lag is directly affected by different workloads. The lowest consumer lag occurs at the undersaturated load, with an average of 0.33 seconds for priority events and 0.30 seconds for normal events. This suggests that the consumers can handle the flow of events efficiently, without much divergence. The fluctuating workload results in higher mean values of 0.50 seconds for priority events and 0.60 seconds for normal events. This increase is because the system sometimes reaches its capacity, but then the flow of events also decreases. The highest consumer

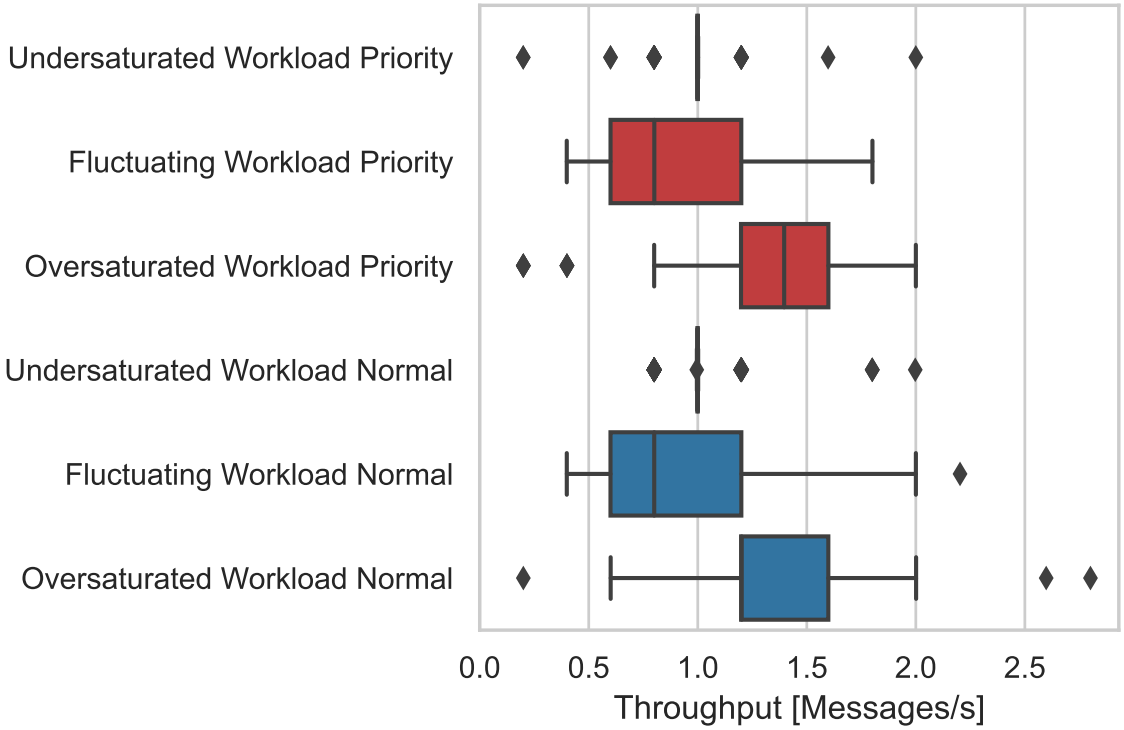


Figure 6.8: The average message throughput for priority events and normal events over time for three workloads: undersaturated, fluctuating, and oversaturated.

lag is found for the oversaturated workload, with a mean of 1 second for both priority and normal events. This indicates that Hestia can balance performance and consistency depending on the flow of events: it prioritises performance when the workload is high, and consistency when the workload is low. Moreover, we notice another significant difference in the consumer lag between priority events and normal events. In all cases, the distribution of consumer lag is wider for normal events. This implies that normal events have less strict consistency requirements. This demonstrates that Hestia can distinguish which events should be prioritised.

Figure 6.8 shows the message throughput per second for the different workloads tested on our prototype. First, we examine the undersaturated workload, which has a very stable throughput, with a mean of around 1. This workload produces and sends a steady stream of events to the broker. The events are processed quickly at the broker, since the rate is set to avoid overwhelming the consumer. However, sometimes the consumer polls the broker when there are no events, which lowers the throughput. Therefore, this workload does not achieve the best throughput, but demonstrates how the prototype operates under

normal circumstances. Next, we look at the fluctuating workload, which has the most variable throughput, ranging from 0.5 to 2.0. This reflects the nature of the workload, which alternates between high and low demand for events. This workload has the lowest throughput because it causes frequent changes in the broker's buffer size, which affects the system performance. Moreover, when the demand is high, the consumer may not be able to keep up with the event rate, leading to longer queues and delays at the broker. On the other hand, when the demand is low, there are no events to consume, which reduces the message throughput drastically. Therefore, both the low and high demand phases of the fluctuating workload contribute to its low throughput. Finally, we consider the oversaturated workload, which has the highest throughput, reaching up to 2.0. This shows the maximum performance of the prototype when it is constantly processing events. This corresponds to how Dyconits is designed: we adjust the consistency bounds based on the message throughput. When the message throughput is high, the policy increases the consistency bounds. As a result, we temporarily have a lower level of consistency, but more events can be processed, so the broker is not overwhelmed. This is also why normal events have a slightly higher throughput in all workloads compared to priority events. The consistency bounds for normal events have less strict consistency requirements, allowing them to be more inconsistent, as shown in Figure 6.7. The trade-off is that these events have a higher throughput and therefore better performance.

Figure 6.9 shows the dyconit messages per second. These are the messages that consumers send to each other with data when they need to synchronise their states. This figure reveals a striking correlation with the results of Figure 6.7 and Figure 6.8. Namely, we see that the dyconit related throughput for the undersaturated workload is the highest. This supports the claim that dyconits offer stronger consistency when the rate of events coming in is low. The consistency bounds can be smaller without losing much in terms of performance. Conversely, we see that in the oversaturated workload, the throughput on dyconit messages is lowest. This indicates that when the inflow of events is high, the consistency bounds will scale more. Additionally, we see that there is a clear difference between how we handle priority events and normal events. Thus, we see that even with an oversaturated workload, the system does its best to keep the priority events better in sync, something that is also evident in Figure 6.7.

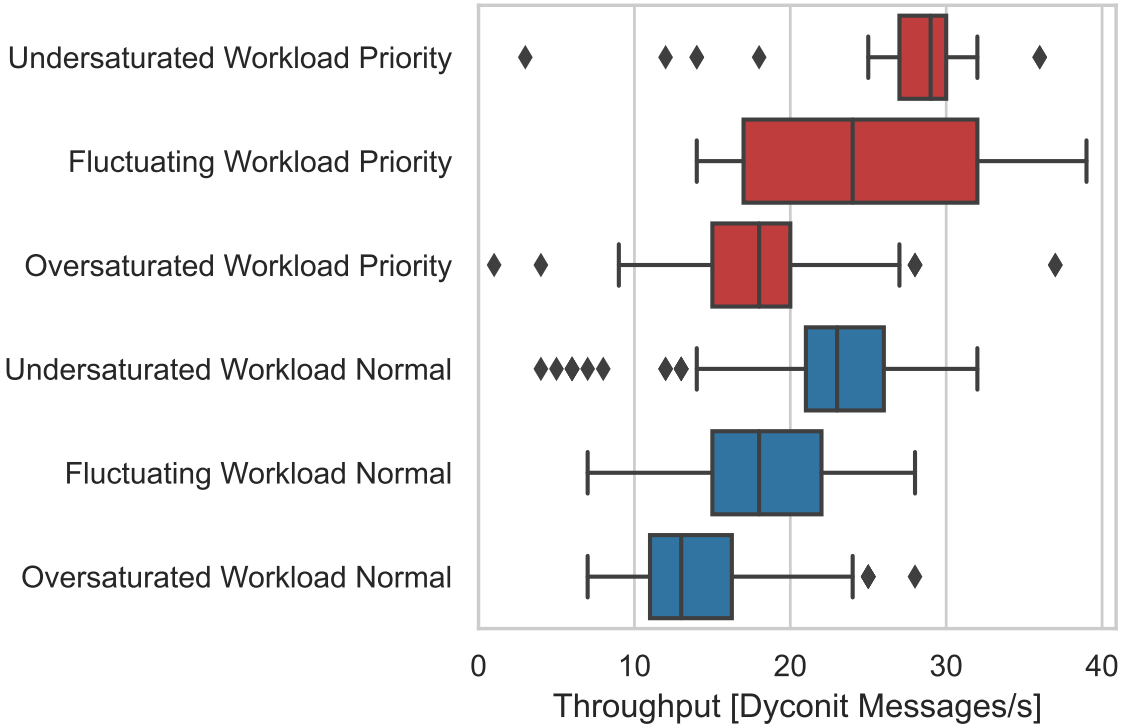


Figure 6.9: The average number of messages exchanged between consumers to synchronise their states over time for priority events and normal events under three workloads: undersaturated, fluctuating, and oversaturated.

6.7.4 Performance Analysis of Hestia Under Varied Policies

MF8 The choice of policy affects Hestia's behaviour significantly, as different policies entail a trade-off between performance and consistency. For example, the Exponential Smoothing Policy can achieve an 81.25% reduction in overhead by allowing a 120% increase in consumer lag.

MF9 Hestia is a flexible and customizable system that can adapt to different applications and user preferences. It offers a variety of policies that can be configured according to the specific requirements and objectives of each scenario.

Figure 6.10 presents a comprehensive view of the consumer lag distribution during the processing of priority and normal events. These events are subjected to distinct policies within the context of an oversaturated workload. We find noteworthy insights into the

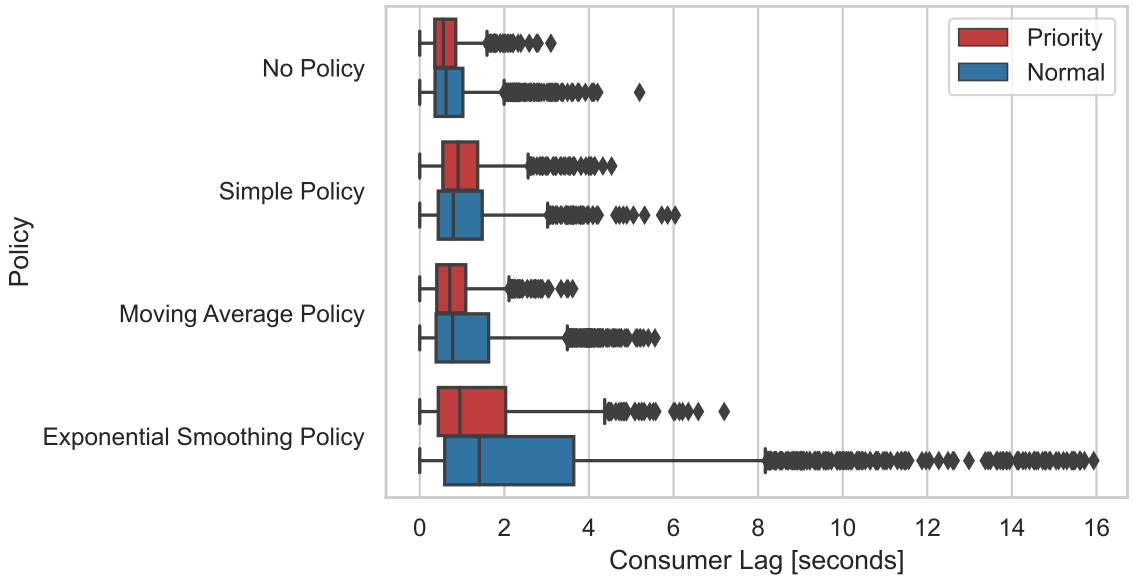


Figure 6.10: Distribution of consumer lag for priority and normal events with different policies using the oversaturated workload.

effectiveness of each policy.

Remarkably, the No Policy approach emerges as the frontrunner in terms of consumer lag performance for both priority and normal events, exhibiting mean values of 0.63 seconds and 0.76 seconds, respectively. This outcome aligns with expectations, given that the absence of a policy preserves the initially stringent bounds. These bounds, being deliberately conservative, contribute to the observed efficiency.

In contrast, the Simple and Moving Average Policies exhibit similar trends, but the latter showcases superior prioritization of priority events. The Simple policy has a 3.9% higher mean consumer lag for priority events than for normal events, but the normal events have a larger standard deviation of consumer lag. Similarly, the Moving Average policy yields mean consumer lags of 0.85 and 1.15 seconds for priority and normal events, respectively. This indicates the Moving Average policy's efficacy in prioritizing priority events within an oversaturated workload, potentially attributable to its more advanced nature.

However, the Exponential Smoothing Policy's results warrant scrutiny. This policy yields a mean consumer lag of 1.39 seconds for priority events and 2.5 seconds for normal events. This divergence in consumer lag indicates the policy's inclination towards managing over-

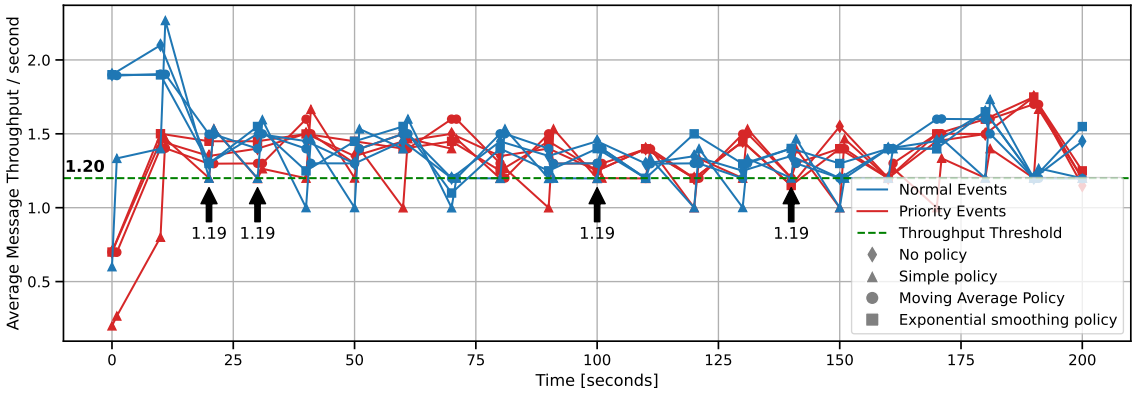


Figure 6.11: Message throughput of various policies under an oversaturated workload, highlighting the significance of threshold choice on system performance and consistency.

saturation by substantially expanding the bounds. As a result, this shift towards larger bounds also translates to extended and less predictable consumer lag following each synchronisation event.

In summary, the analysis underscores that the No Policy approach capitalizes on conservative bounds to maintain the lowest consumer lag across both priority and normal events. Conversely, the Exponential Smoothing Policy's trade-off between expanded bounds and increased consumer lag illustrates the complexity of optimizing performance in oversaturated workloads.

In Figure 6.11 we analyse the message throughput of different policies under the oversaturated workload. there are no large differences between the policies in terms of message throughput. However, a notable observation is that the throughput often exceeds the threshold of 1.20 messages per second for all the policies. This indicates that the bounds are increased more frequently for all the policies to handle the over saturation of the workload. This is consistent with the nature of the workload, which pushes the system to the limit. Another interesting observation is that in some cases, the throughput is slightly below the threshold of 1.20 messages per second. In these cases, the bounds will decrease. This indicates that the choice of the threshold can have a significant impact on the consistency and performance of the system. We further analyse this in Experiment 6.6.5.

Figure 6.12 provides an explanation for the results in the previous graphs. First, we see that for all policies, there is more overhead for the priority events compared to the normal

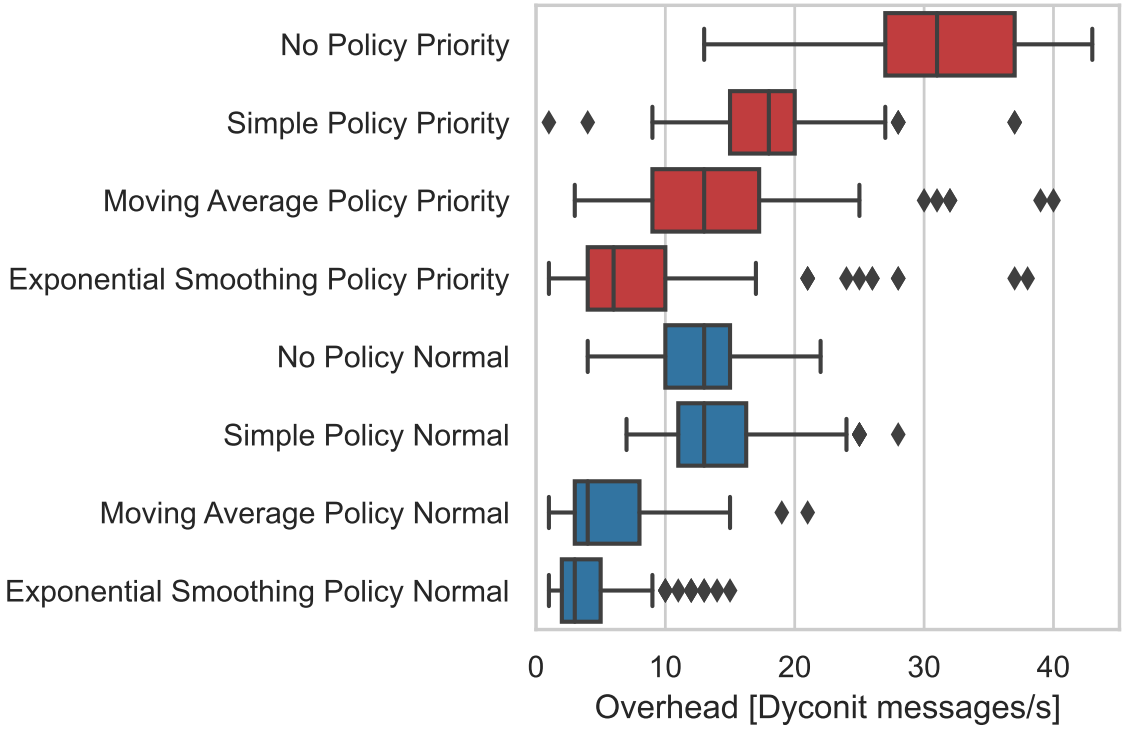


Figure 6.12: Overhead comparison of different policies for priority versus normal events.

events. This is an expected result, as we have indicated that these events are more important and need a higher degree of consistency. However, we see that we have by far the most overhead when we do not use a policy. As indicated earlier, the initial bounds are quite strict, so a lot of synchronisation is needed. On the one hand, this makes for a very consistent system; on the other hand, you do sacrifice some performance in return. Furthermore, we can see from this figure that the more specialized the policy is, the more impact it has on the performance. For example, the exponential smoothing policy is configured to increase the bounds to provide more events. Therefore, it requires less communication between services. In contrast, a simpler policy like simple policy still loses quite a lot of performance. Ultimately, the choice of policy depends on the software engineer's preference for performance or consistency, as well as their ability to develop their own policies. This is why it is important for software engineers to also have the freedom to create their own policies.

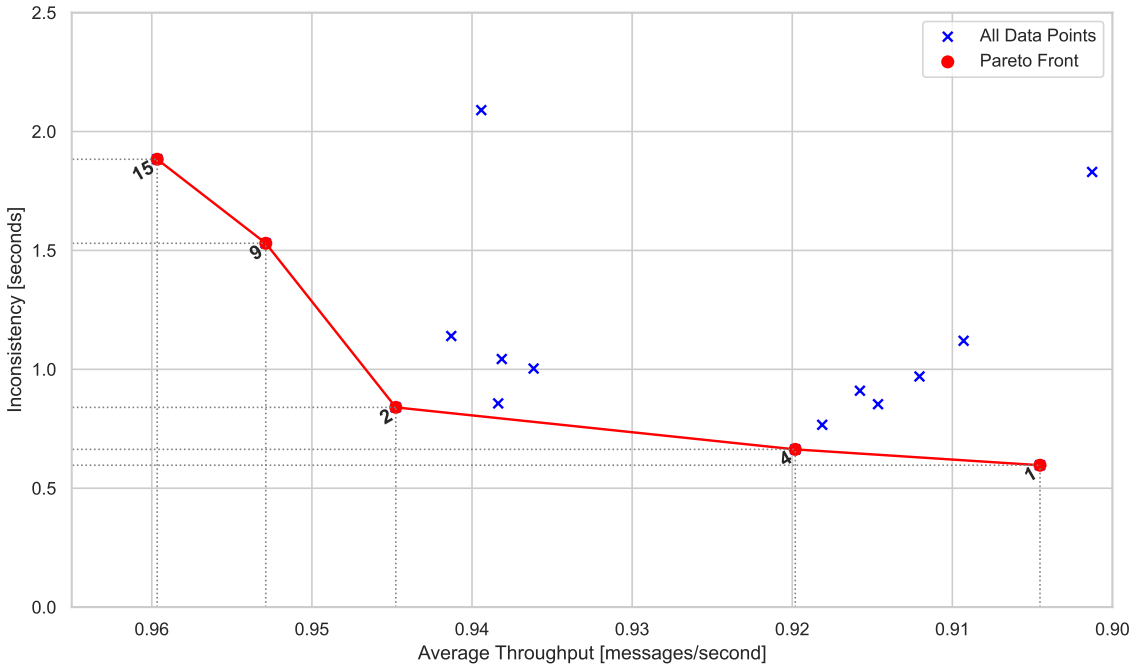


Figure 6.13: Pareto front representation for priority events under a fluctuating workload. The axes depict maximum inconsistency lag and average event throughput The Pareto front visualizes the optimal trade-offs between these objectives, highlighting settings that offer the best balance between minimizing inconsistency and overhead while maximizing message throughput.

6.7.5 Sensitivity Analysis of Varied Dependent Policy Parameters

MF10 Factors such as initial bounds, policy rules, and thresholds affect the system's performance in terms of consistency, overhead, and throughput. For example, the configuration that gives priority events the most flexibility shows a 70% higher inconsistency rate than the configuration that ensures high consistency, but it also reduces the overhead by about 60% compared to the configuration that imposes the most strictness.

MF11 The best system configuration varies depending on the specific scenario and the objectives and preferences of the software engineer. There is no universal solution; rather, there is a set of optimal trade-offs for different situations, which can be represented by a Pareto frontier. Generally speaking, we observe that tight bounds offer a balance of weak consistency and high performance, while loose bounds ensure strong consistency but lower performance.

Figure 6.13 shows a Pareto front for priority events in a fluctuating workload. The two axes represent the maximum inconsistency lag (how far a service lags behind another ser-

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	$+8, -4$

vice) and the average event throughput (how many events the consumer can process per second). A Pareto front is a graphical representation of the optimal trade-offs between multiple objectives in a decision-making problem. It shows the set of solutions that are Pareto efficient, meaning that no other solution can improve one objective without worsening another. Our goal was to minimise the inconsistency, while maximising the message throughput. Table 6.8 lists the most optimal settings along with their corresponding lag and throughput values, as well as the threshold and the rules used for each setting. The threshold number is based on throughput, and the rules are the actions taken when the throughput exceeds or falls below the threshold. For example, setting 2 has a threshold of 1.0 and rules of $+1, -.5$, which means that if the throughput exceeds 1.0, the consumer will increase both the staleness bound (the maximum allowed difference between two services) and the numerical error (the maximum allowed deviation from the exact value) by 1 second, and if it is lower than 1.0, it will decrease them by 0.5 seconds.

We can use both the table and the figure to determine what configuration is optimal for an event-driven system that uses priority events. We see that there is a trade-off between low inconsistency and low message throughput on one hand, and high overhead on the other hand. This is consistent with previous experiments that we conducted. However, on the opposite side of the Pareto front, we see that the highest inconsistency also yields the highest message throughput and the lowest overhead throughput. Therefore, this experiment shows that using very conservative rules for the policies yields very strict consistency results. This is because the initial boundaries were already quite strict, so changing them slightly when the throughput varies does not affect them much. As a result, when using rules that adjust their bounds more significantly, we see that we are on the other side of our Pareto front. Those bounds have a higher impact in changing the initial strict bounds. We

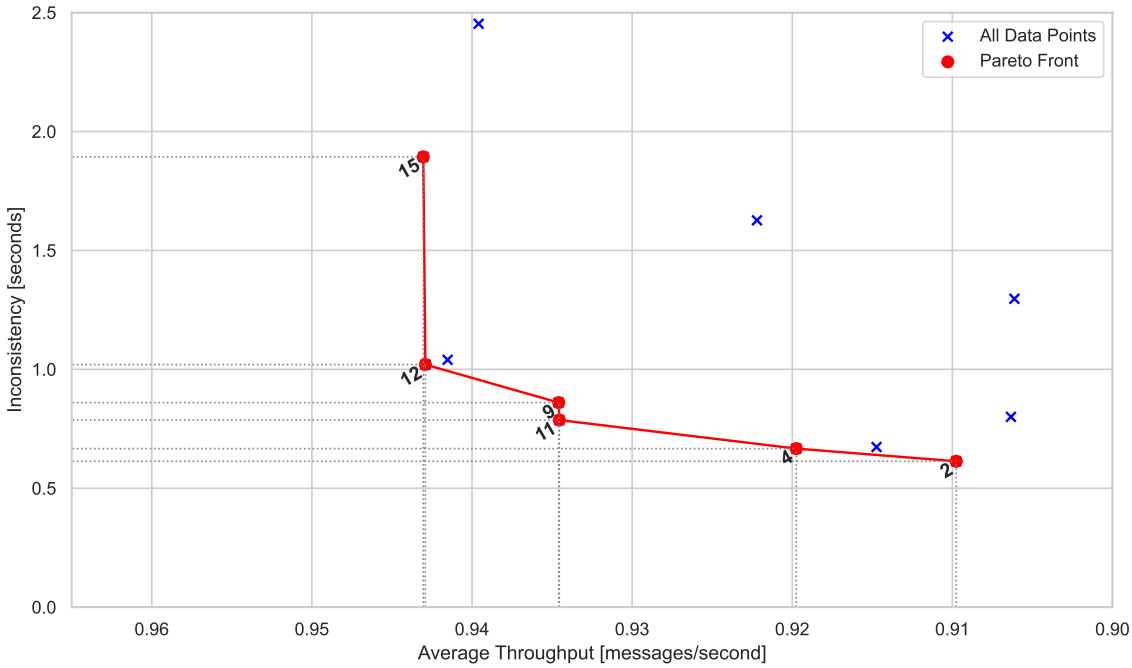


Figure 6.14: Pareto front representation for normal events under a fluctuating workload. The axes depict maximum inconsistency lag, and average message throughput. The Pareto front visualizes the optimal trade-offs between these objectives, highlighting settings that offer the best balance between minimizing inconsistency and overhead while maximizing message throughput.

see that generally this results in weaker consistency regardless of the throughput threshold. This means that throughput has been above the threshold more often than below it, which increased bounds but kept them within limits.

One conclusion that can be drawn from these results is that the optimal configuration for an event-driven system that uses priority events depends on the trade-off between consistency and performance. If the system requires strict consistency, then it should use conservative rules that keep the bounds tight and accept the cost of high overhead. If the system can tolerate some inconsistency, then it should use aggressive rules that loosen the bounds and benefit from high message throughput and low overhead. The Pareto front provides a visual guide to choose the best configuration based on the desired objectives. Figure 6.14 shows the Pareto front for normal events in a fluctuating workload. Our goal is the same as for the priority events: minimise inconsistency and overhead while achieving maximum throughput. The most optimal configurations and their corresponding values are shown in Table 6.9. The results reveal that the normal events have a lower priority in

Table 6.9: Optimal settings for normal 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
1	596	0.904526	8.375	0.8	$+1, -.5$
4	663	0.919813	8.343	1.4	$+1, -.5$
10	856	0.938366	6.000	1.0	$+4, -2$
12	1140	0.941297	5.134	1.4	$+4, -2$
9	1530	0.952880	4.838	0.8	$+4, -2$
15	1883	0.959662	4.210	1.2	$+8, -4$

synchronising with other services than the priority events, as they have a significantly lower overhead with similar lag and message throughput. This implies that the normal events can tolerate more inconsistency for longer periods of time. Another observation is that the same conclusion that we drew for the priority events also holds for the normal events: the conservative rules generally provide the strongest consistency, while the aggregate bounds offer the highest throughput and the lowest overhead at the expense of consistency.
The conclusion that we can draw from the results is that the performance of the system depends on the initial bounds, the policy rules, and the threshold, as well as the type and priority of the events. The initial bounds determine the range of possible values for consistency, overhead, and throughput. The policy rules and the threshold control how the system adapts to the fluctuating workload and how it balances the trade-offs between the three metrics. The type and priority of the events affect how much inconsistency and overhead the system can tolerate and how much throughput it can achieve.

To summarise, we can say that there is no one-size-fits-all solution for the system configuration, but rather a Pareto front that shows the optimal trade-offs for different scenarios. Software Engineers should carefully choose the initial bounds, the policy rules, and the threshold according to their specific needs and preferences. They should also consider the characteristics and priorities of the events that they expect to handle in their system. By doing so, they can optimise the performance of their system and achieve their desired goals.

6.8 Discussion

Due to the challenges setting up a realistic environment with multiple nodes and scaled workloads, we did not test our system on a real setup. However, we believe that our results

are indicative of the potential benefits of Dyconits in such scenarios. Based on existing benchmarks, we expect that Kafka would exhibit similar performance characteristics as in our experiments, but with higher latency and lower throughput due to network overhead and contention (43). We also assume that Hestia would be able to adapt to the changing conditions and maintain the desired consistency levels for different event priorities. However, we acknowledge that these are speculative claims and that there might be unforeseen factors or challenges that could affect the behaviour and performance of our system in a real setup. Therefore, we plan to conduct further experiments on a real setup in the future to validate our findings and address any limitations or issues that might arise.

6. EVALUATION OF THE PERFORMANCE OF A GENERIC DYCONIT SYSTEM: EXPERIMENTAL DESIGN AND RESULTS

7

Conclusion and Future Work

This chapter summarizes the contributions of this thesis and looks forward to future lines of research emerging from it.

7.1 Conclusion

In this work, our goal was to extend Dyconits to general event-driven systems. To achieve this, we have formulated four main research questions that guide our investigation, as presented in Chapter 1. We have also reviewed the relevant literature on Dyconits and related topics in Chapter 2. In Chapters 3 to 6, we have described our contributions in terms of requirements analysis, design, implementation, and evaluation of our proposed approach. In this section, we revisit our research questions and provide answers based on our findings.

(RQ1) State-of-the-art: What are the common requirements in event-driven systems that can inform the design of a generic Dyconit system?

we present a universal topology for event-driven architectures that can support various communication patterns and application contexts. We then evaluate the benefits and drawbacks of each communication pattern and application context, and examine how well our universal topology addresses the diverse demands and challenges of event-driven architectures. Moreover, we illustrate its generality by applying it to three different applications that involve synchronizing replicated data in the presence of service latency. These applications are smart farming, chat application, and real-time analysis. This allows us to identify their specific non-functional requirements.

(RQ2) Design of the system: How to design a generic Dyconit system for event-driven systems?

Our requirement analysis of event-driven systems led us to synthesize the essential functional and non-functional requirements for a generic Dyconit system. The Dyconit system design for event-driven architectures has three main components: the Dyconit Overlord, the Dyconit Admin, and the Dyconit Agents. These components are responsible for managing the dyconits and their consistency policies. Our design for the generic Dyconit consistency model belongs to the application-oriented perspective, under the dynamic/optimistic subcategory. This implies that the system can provide different consistency levels for different dyconits based on their importance and relevance.

(RQ3) Implementation of the system: How to integrate a generic Dyconit system into event-driven systems?

We present a working prototype of the generic dyconit system, called Hestia, that fulfils the design's functional and non-functional requirements. We use the Confluent.Kafka library for .NET, which provides a rich set of APIs and abstractions for interacting with Apache Kafka. This library enables us to create and configure producers and consumers, as well as applications that can perform both roles. Moreover, we develop custom components to handle the creation, update, and deletion of dyconits.

(RQ4) Evaluation of the system: How to evaluate a generic dyconit system for event-driven system?

We present Hestia, a prototype for optimistic inconsistency in event-driven systems, based on the dyconit consistency model. We evaluate Hestia with synthetic workloads, two common topologies, and different policies, and compare it with a baseline system. Hestia reduces inconsistency by 70% for both priority levels, while decreasing the throughput by 45%. Hestia adapts to the workload saturation by balancing performance and consistency. Hestia is flexible and customizable to different applications and user preferences. Hestia can also prioritize certain events according to the application's needs. We conclude that Hestia offers a useful trade-off between inconsistency, throughput, and overhead, and that software engineers can tailor them to their specific requirements. In short, Hestia can significantly reduce inconsistency, but with some overhead and lower throughput.

7.2 Future Work

Our prototype leverages the idea of application-centric consistency to enable services to retrieve more recent data from faster services at similar positions in the topology. Application-centric consistency allows applications to specify their own consistency policies and preferences, rather than relying on the default or a one-size-fits-all approach of the underlying system. This gives more flexibility and control to the application developers and users, and enables them to optimize their data access and processing according to their specific needs and scenarios.

In our prototype, we use staleness and numerical error bounds as the parameters for bounding inconsistency. By using these parameters, services can express their tolerance for data freshness, and request data from other services that can satisfy their requirements. We implement this idea in our prototype system, Hestia, which is a middleware layer that supports application-centric consistency for event-driven systems. Dyconits enables services to retrieve more recent data from faster services at similar positions in the topology.

Our work demonstrates the potential for improving the quality attributes of event-driven systems, especially in dynamic and heterogeneous environments. We envision event-driven systems that can handle large-scale and complex data streams with high efficiency and accuracy. We aim to advance this exciting and important field of research. We invite other researchers and practitioners to explore new ways of using Dyconits and application-centric consistency in event-driven systems. As future work, we suggest 4 directions, building on the contributions of this thesis:

1. We propose that *application-centric consistency models are a promising direction for future research* in event-driven systems. We demonstrate this by presenting Hestia, our prototype system that allows fine-grained control over the consistency level of different types of events. We argue that this approach is more suitable for event-driven systems than traditional consistency models, as it can better accommodate the diverse and dynamic nature of events. Therefore, we encourage the research community to explore further the potential and challenges of application-centric consistency models.

2. Other work can *deepen and engineer Hestia to be fit for deployment in a production context.* Hestia can be optimized to reduce the overhead of managing dyconits,

such as minimizing the communication cost between the overlord and the admins, and handling failures and recovery. Additionally, we challenge the community to look further into what other implementations work for a dyconit system in event-driven systems. Earlier versions of the design we have shown that there is promising potential developing a system that not only looks at the consumers, but also considers the whole system. Namely, the producers, broker, and consumer. One possible idea is to leverage the information available at the broker level. For example, in Kafka, the API provides information about how far behind the consumer is from the producer. This information can be used to adjust the rate of event production and consumption dynamically, depending on the consumer's load and availability. This way, the system can balance the trade-off between consistency and performance by adapting to the changing conditions of the event-driven system.

3. We challenge the community to *implement and evaluate Hestia in a large-scale distributed setting*, and to investigate how to optimize the consistency level for each event type, how to evaluate the trade-offs between consistency and performance. Our experiments on Hestia have the limitation that they are executed on a single machine docker swarm setup. This gave us the advantages of a controlled environment, simplicity in configuration, and quick iteration times, but also presented disadvantages such as limited scalability, non-representation of real-world network latencies, and the inability to assess inter-service communication challenges. Therefore, a comprehensive evaluation in a more complex and distributed infrastructure would offer invaluable insights into Hestia's capabilities and areas for improvement.

4. Expanding on future advancements in the dyconit consistency model, we suggest *harnessing the capabilities of Machine Learning and Artificial Intelligence for defining consistency bounds and formulating policies.* These technologies offer enhanced potential to navigate the vast design space associated with these parameters. Although the task will likely continue to be human-centric, leveraging an AI tool can significantly alleviate the burden on practitioners. Multi-objective optimization algorithms, prevalent in both traditional Artificial Intelligence and contemporary Machine Learning methodologies, warrant specific evaluation for this challenge. Conducting a user study that contrasts selected methodologies with our proposed ones would offer a robust measure of their effectiveness.

8

Appendix

8.1 Email Questions to Industry Experts

E-mail

I am currently working on my thesis around the area of consistency. I am currently working on developing a generic consistency model that provides "optimistic consistency". My approach to realising this consistency model is to implement it as a wrapper on Kafka.

I would like to test my implementation soon by using different "real-world" patterns of events added to Kafka. Through CONTACT, I understand that you have experience with Kafka at COMPANY. Therefore, I'm curious if you can share with me what different workloads you see coming in.

Specifically, I'm interested in the following details:
Q1: The average number of events you process per second/minute
Q2: The pattern of these events. For example, are there peak moments at certain times?
Q3: The average size of these events (in kilobytes or megabytes)
Q4: Is there a difference in the importance of events, e.g. between logging events and other types of events?

By understanding the workload of 'real-world applications', I can better tune and optimise my system.

Thanks in advance for your time and effort.

Respondent 1

The first respondent is the manager of DataRotonde. DataRotonde is a service that helps housing corporations manage IT incidents and coordinate with external vendors to resolve issues in their application connections, ensuring faster and proactive problem-solving.

A1:
“Looking at the last six hours, across all tenants, it averaged 1,834 / minute or 30.5 / second. The busiest minute was at 4,457 / minute. This is what we get in terms of production messages on the various engine instances. Logging events are not part of this. Logging (run via RabbitMQ) averaged 24,062 / minute or 401 / second over the same period.”

A2:
“Yes, traffic is fairly dependent on working hours at connected parties. In addition, data changes can cause extensive data synchronisations that generate many events. The traffic is so diverse that no more specific patterns can be found in it, which varies by connected process and organisation.”

A3:
“The average size over the same period is 2938 bytes. But the size of the events follows a lognormal distribution with a very long tail. The median is 101 bytes on average.”

A4:
“Not really, logging events are needed for the audit trails and we also want to keep them as complete as possible. Should the need arise, we could differentiate between dropping certain types of events. DataRotonde is not currently set up for that.”

Respondent 2

The second respondent is a consultant working for a company in the transport sector.

A1:
The average number of events processed per second/minute is 300 during peak moments, which occur during the morning and late afternoon rush hours in the transport sector.

A2:
The workload exhibits peaks between 06:00 - 09:00 and between 15:00 - 18:00, corresponding to the rush hour periods. These are the times when the highest number of events are generated and need to be processed.

A3:
The average size of events in the transport sector is 20 kilobytes (KB), with some large outliers. Over the course of the last month, there were 73.7 million rows of data, amounting to a total of 382.5 gigabytes (GB) of data.

A4:
In the transport sector, all events are considered equally important. There is no differentiation in terms of event importance, as every event holds significance for the operations and management of the transport systems.

Respondent 3

The third respondent is a consultant working in the energy sector.

A1:
“Unfortunately, the specific average number of events processed per second/minute is not provided by the third respondent in the energy sector.”

A2:
“The workload in the energy sector experiences peak moments. However, the exact timing or frequency of these peak moments is not mentioned, so further details regarding the specific patterns are not available.”

A3:
“The events in the energy sector tend to have large sizes. In order to handle these large events effectively, they are divided into thousands of smaller events. Additionally, the third respondent mentions that large events are relatively rare, implying that the majority of events are smaller in size.”

A4:
“In the energy sector, separate topics are utilized for different priority events. This suggests that there is a distinction made between various types of events based on their importance or priority. The system is designed to handle different types of events and prioritize them accordingly using separate topics.”

Respondent 4

The fourth respondent is a software engineer working at a retail e-commerce company.

A1:
“On average, we process around 50 events per second and approximately 3000 events per minute.”

A2:
“Our workload exhibits distinct patterns throughout the day. We observe peak moments during weekdays between 10:00 AM and 2:00 PM, coinciding with high customer activity and order placements. We also experience increased traffic during promotional campaigns and holiday seasons.”

A3:
“The average size of our events varies depending on the nature of the data being processed. Order-related events, which include customer information and purchased items, range from 10 to 50 kilobytes (KB). Inventory update events, such as stock level changes, are smaller, averaging around 1 kilobyte. However, media-related events, such as product images or videos, can be larger, ranging from a few hundred kilobytes to several megabytes.”

A4:
“In our system, there is a difference in the importance of events. Order-related events are considered crucial for order processing and fulfillment, as they directly impact customer satisfaction. Inventory update events are also important to ensure accurate stock information. However, generic system logs or internal monitoring events have a lower priority in terms of processing and storage, as they don't directly impact customer experience or business operations.”

Respondent 5

The fifth respondent is a consultant working at a financial institution.

A1:
“Our event processing rate is relatively low due to the high sensitivity and criticality of the data. On average, we handle around 10 events per second.”

A2:
“Our workload doesn't exhibit significant peaks or specific patterns in terms of volume or timing. The events are spread out evenly throughout the day incidents can occur at any time.”

A3:
“The size of our events is relatively small, typically ranging from a few hundred bytes to a few kilobytes. The events primarily consist of logs and alerts.”

A4:
“In our system, all events are considered equally important.”

8.2 Policy Configurations

```
{
  "policy": "throughput-threshold-policy",
  "collectionNames": ["topic_priority", "topic_normal"],
  "thresholds": {
    "throughput": 5,
    "overhead_throughput": 4
  },
  "rules": [
    {
      "policyType": "standard",
      "condition": "throughput >= threshold",
      "actions": [
        {
          "type": "multiply",
          "value": 0.85
        }
      ]
    },
    {
      "policyType": "standard",
      "condition": "throughput < threshold",
      "actions": [
        {
          "type": "multiply",
          "value": 1.15
        }
      ]
    }
  ]
}
```

Listing 2: Simple Policy

```
{
"policy": "moving-average-policy",
"collectionNames": ["topic_priority", "topic_normal"],
"averageSizeThroughput" : 3,
"thresholds": {
  "throughput": 5
},
"rules": [
  {
    "policyType": "standard",
    "condition": "avg > threshold",
    "actions": [
      {
        "type": "multiply",
        "value": 1.05
      }
    ]
  },
  {
    "policyType": "standard",
    "condition": "avg < threshold",
    "actions": [
      {
        "type": "multiply",
        "value": 0.95
      }
    ]
  }
]
}
```

Listing 3: Moving Average Policy

```
{
"policy": "expsmoothingPolicy",
"collectionNames": ["topic_priority", "topic_normal"],
"thresholds": {
  "throughput": 5
},
"rules": [
  {
    "policyType": "exponentialSmoothing",
    "smoothingFactor": 0.5,
    "condition": "throughput > threshold",
    "actions": [
      {
        "type": "multiply",
        "value": 1.05
      }
    ]
  },
  {
    "policyType": "exponentialSmoothing",
    "smoothingFactor": 0.5,
    "condition": "throughput < threshold",
    "actions": [
      {
        "type": "multiply",
        "value": 0.95
      }
    ]
  }
]
}
```

Listing 4: Exponential Smoothing Policy

References

[1] Hesam Nejati Sharif Aldin, Hossein Deldari, Mohammad Hossein Moattar, and Mostafa Razavi Ghods. **Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications**. *CoRR*, **abs/1902.03305**, 2019. 1

[2] Susanne Braun, Stefan Dessloch, Eberhard Wolff, Frank Elberzhager, and Andreas Jedlitschka. **Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems - An Action Research Study**. *CoRR*, **abs/2108.03758**, 2021. 1

[3] Paolo Viotti and Marko Vukolić. **Consistency in Non-Transactional Distributed Storage Systems**. *CoRR*, **abs/1512.00168**, 2015. 1, 2, 10

[4] Maurice P. Herlihy and Jeannette M. Wing. **Linearizability: A Correctness Condition for Concurrent Objects**. *ACM Trans. Program. Lang. Syst.*, **12**(3):463–492, jul 1990. 1

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. **Impossibility of Distributed Consensus with One Faulty Process**. *J. ACM*, **32**(2):374–382, apr 1985. 1

[6] Douglas Terry, Alan Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. **Session guarantees for weakly consistent replicated data**. pages 140 – 149, 10 1994. 2, 13

[7] Dedy Kristiadi, Ferry Sudarto, Evan Rahardja, Naufal Hafizh, Christopher Samuel, and Harco Leslie Hendric Spits Warnars. **Mobile cloud game in high performance computing environment**. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, **18**:1983, 08 2020. 2

[8] Maarten van Steen and Andrew S. Tanenbaum. **A brief introduction to distributed systems**. *Computing*, **98**:967–1009, 2016. 2, 12

[9] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 126–137, 2021. 2, 19, 22, 48

[10] Haifeng Yu and Amin Vahdat. **Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services**. *ACM Trans. Comput. Syst.*, **20**(3):239–282, aug 2002. 2, 16, 17, 48

[11] Jurre Brandsen, Jesse Donkervliet, and Alexandru Iosup. **A Comprehensive View Of Consistency in Distributed Systems**. 2023. 3, 9, 13

[12] G. Licoppe and F. Deraedt. *Calepinus novus: vocabulaire latin d'aujourd'hui*. Fondation Melissa et Musée de la Maison d'Erasme, 2002. 9

[13] David Bermbach and Jörn Kuhlenkamp. **Consistency in Distributed Storage Systems**. In Vincent Gramoli and Rachid Guerraoui, editors, *Networked Systems*, pages 175–189, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 9, 10

[14] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. **Analyzing Consistency Properties for Fun and Profit**. PODC '11, page 197–206, New York, NY, USA, 2011. Association for Computing Machinery. 10

[15] Seth Gilbert and Nancy Lynch. **Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services**. *SIGACT News*, **33**(2):51–59, jun 2002. 10

[16] Daniel Abadi. **Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story**. *Computer*, **45**(2):37–42, 2012. 10

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1992. 10

[18] Edsger W. Dijkstra. *Cooperating Sequential Processes*, pages 65–138. Springer New York, New York, NY, 2002. 11

[19] Leslie Lamport. **Time, Clocks and the Ordering of Events in a Distributed System**. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, pages 558–565, July 1978. 11

[20] P.W. Hutto and M. Ahamad. **Slow memory: weakening consistency to enhance concurrency in distributed shared memories**. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 302–309, 1990. 12

[21] Werner Vogels. **Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs?Between Consistency and Availability.** *Queue*, **6**(6):14–19, oct 2008. 12

[22] Eric Brewer. **Towards robust distributed systems**. page 7, 07 2000. 12

[23] Seth Gilbert and Nancy Lynch. **Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services**. *SIGACT News*, **33**(2):51–59, jun 2002. 12

[24] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. **Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary**. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 265–278, USA, 2012. USENIX Association. 12

[25] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. **A Commutative Replicated Data Type for Cooperative Editing**. *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, 2009. 12

[26] J. Donkervliet. **Design and Experimental Evaluation of a System based on Dynamic Conits for Scaling Minecraft-like Environments**. 2018. 19

[27] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems**. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020. 19

[28] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. **Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games**. *SIGCOMM Comput. Commun. Rev.*, **38**(4):389–400, aug 2008. 19

[29] Michael Macedonia, Michael Zyda, David Pratt, Paul Barham, and Steven Zeswitz. **Npsnet: A Network Software Architecture For Large Scale Virtual Environments**. *Presence*, **3**:265–287, 01 1994. 19

[30] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent van Beek, and Lucian Toader. **Massivizing Computer Systems: A Vision to Understand, Design, and Engineer Computer Ecosystems Through and Beyond Modern Distributed Systems**. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1224–1237, 2018. 22

[31] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004. 23

[32] Jianwei Zhang and Antonio Vallecillo. **Realizing Model-Driven Web Engineering with Event-Driven Architecture**. *Journal of Web Engineering*, **13**(56):383–409, 2014. 23

[33] Zohaib Mehdi, Naveed Ahmed, and Babar Shahzad. **Scalability in event-driven architecture**. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018. 23

[34] Jun Narkhede, Neha Shapira, and Jay Jain. **Apache Kafka: A Distributed Streaming Platform**. *Confluent Inc.*, 2017. 26

[35] Apache Kafka. **Semantics - Apache Kafka**. `https://kafka.apache.org/documentation/#semantics`, 2012. Accessed May 23, 2023. 27, 48

[36] Julien Kervizic. **Real-time Data Pipelines — Complexities Considerations**, December 2020. Accessed: 2023-05-25. 31

[37] Michael Stonebraker, Uundefinedur Çetintemel, and Stan Zdonik. **The 8 Requirements of Real-Time Stream Processing**. *SIGMOD Rec.*, **34**(4):42–47, dec 2005. 32

[38] S.D. Kuznetsov, P.E. Velikhov, and Q. Fu. **Real-Time Analytics: Benefits, Limitations, and Tradeoffs**. *Program Comput Soft*, **49**(1):1–25, 2023. 32

[39] Douglas C. Schmidt and Carlos O'Ryan. **Patterns and performance of distributed real-time and embedded publisher/subscriber architectures**. *Journal of Systems and Software*, **66**(3):213–223, 2003. Software architecture – Engineering quality attributes. 33

[40] Microsoft. **Publisher-Subscriber pattern - Azure Architecture Center | Microsoft Learn**, December 2022. Accessed: 2023-05-16. 33

[41] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776, 2019. 39, 43, 54

[42] Jesse Donkervliet. **Design and Experimental Evaluation of a System based on Dynamic Conits for Scaling Minecraft-like Environments**. 2018. Accessed: 2023-05-25. 49

[43] Jay Keps. **Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)**. Accessed: 2023-08-18. 105