Vrije Universiteit Amsterdam

Bachelor Thesis

Dynamically Managed Inconsistency in Distributed Systems

A Dyconit Middleware for Publish-Subscribe Systems

Author: Martin Karsai (2737119)

1st supervisor: ir. Jesse Donkervliet
2nd reader: prof. dr. ir. Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 20, 2024

Abstract

Distributed systems allow online services and applications to scale and distribute workloads effectively. However, maintaining consistency in these systems is a significant challenge. Consistency is crucial for ensuring that distributed systems are reliable and predictable, especially when handling concurrent data access and updates. Recently, innovative approaches have enabled fine-grained dynamic consistency in distributed systems.

This thesis aims to extend dynamic consistency models onto distributed systems following the publish-subscribe messaging pattern. It focuses on Dyconits, a middleware designed to dynamically and optimistically bound consistency in Modifieable Virtual Environments (MVEs) and increase the performance of the system. Dyconits provide promising results in consistency models outside MVEs, but their precise benefits and drawbacks in other environments still need to be researched.

I present a working prototype of Dyconits in a publish-subscribe system. My design and implementation successfully translate the concept of Dyconits and enable fine-grained optimistic and dynamic consistency management. My work leverages the well-known performance and consistency trade-off in distributed systems and maintains the loosely coupled nature of publish-subscribe architecture. My results indicate the potential improvement Dyconits can bring to different environments. In my experiments, I reduced the CPU usage of the underlying system by 30% and decreased the inconsistency of high-priority events by 50% but significantly decreased the consistency of lower-priority topics. Experiments under various circumstances and the highly configurable nature of my prototype provide options to configure my design for the use case-specific needs. My results suggest many possibilities for research into the area of dynamic consistency models in distributed systems.

My functional prototype can be found on GitHub.

Contents

1

Introduction

1.1 Context

Distributed systems are a fundamental architecture in modern computing; for example, Apache Kafka, a distributed event streaming platform, alone is used by more than 80% of Fortune 100 companies (1). Distributed systems enable multiple connected nodes to work together as a single system (2). By distributing workloads and resources across various locations, these systems enhance performance, scalability, and reliability (3). Despite their benefits, distributed systems face significant challenges, including managing data consistency (2).

In the context of distributed systems, consistency refers to a guarantee that all nodes in the system have the same data at a given time (4). Achieving consistency presents a significant challenge due to various factors such as network delays, concurrent updates, partitioning and partial or full system failures (5). Maintaining consistency can come at a considerable performance cost. On the other hand, relaxing consistency requirements can significantly improve performance by allowing nodes to operate independently. However, it can lead to anomalies such as stale or inconsistent data being read. This is a well-recognised trade-off between consistency and performance.

To overcome the challenges and improve the scalability of Modifiable Virtual Environments (MVEs), Donkervliet et al. introduced dynamic consistency units, Dyconits, (6). Dyconits serve as middleware that optimistically and dynamically bounds inconsistency in MVEs and improves their scalability. Dyconits are a continuous consistency model based on a data-centric definition of consistency first proposed in TACT (7). Most importantly, this model allows inconsistencies to be quantified and bounded.

This thesis explores the generalisation of the Dyconits consistency model concept to other distributed systems. It is mainly concerned with the consistency-performance trade-off in distributed publish-subscribe systems. Publish-subscribe is a popular messaging pattern in distributed systems due to its scalability.

1.2 Problem Statement

In the context of publish-subscribe platforms, ensuring data consistency while maintaining scalability and performance is a critical challenge. Current models, such as strict or eventual consistency, sacrifice consistency for performance or vice versa. However, current approaches may not be optimal for every use case.

Recently, Donkervliet et al. have proposed a novel concept for managing consistency, Dyconits. Dyconits have been originally designed and evaluated for dynamically and optimistically managing consistency within MVEs. Their approach to bounding inconsistency while maintaining performance provides a promising area for research in extending the use cases of Dyconit outside of MVEs (6). Due to the large-scale distributed nature of many publish-subscribe systems, a fine-grained dynamic consistency approach provided by Dyconits promises favourable results in terms of consistency management depending on use case-specific requirements.

The premise of my research's societal impact is to create a suitable consistency model for a variety of modern applications. Dyconits' approach has the potential to better use the available resources compared to a one-size-fits-all approach. This could lead to lowered energy consumption or increased performance of current systems.

1.3 Research Questions

This work aims to extend the concept of Dyconits beyond MVEs and apply it to publish-subscribe systems. I hypothesise that using Dyconits to achieve consistency can also benefit these systems' performance and consistency. To assess my hypothesis, I propose the following research questions:

RQ1 Design of the system: How to design a Dyconit system for publish-subscribe systems? So far, the research into the use of Dyconits in MVEs has produced promising results, improving scalability while bounding inconsistency. This suggests a potential for extending their application to other areas where scalability and consistency are crucial. However, designing a Dyconit presents significant

challenges as the concept of Dyconits is still new, and more research is required to comprehend and fully realise its potential across various fields.

RQ2 Implementation of the system: How to integrate a Dyconit system for publish-subscribe systems? Implementing the design is essential to fully extending the concept of Dyconits to other domains. Integration of Dyconits into new domains can pose many challenges, as publish-subscribe systems and distributed systems are often complex and can vary widely between different domains.

RQ3 Evaluation of the system: How to evaluate a Dyconit system for publish-subscribe systems? Dynamic and optimism inconsistency can improve the system's performance by allowing for limited discrepancies in the system's state across different nodes. Evaluating the results of a system designed to leverage the performance and consistency trade-off can be very use-case-specific. The desired results can vary between different environments, setups and configurations. This thesis aims to analyse the ability of a Dyconit system prototype by performing various experiments and looking at relevant metrics.

1.4 Thesis Contributions

Following the specified research questions, my work makes the following contributions.

C1 Design of a Dyconit system for publish-subscribe systems. This design extends the use of Dyconits from MVEs into a publish-subscribe domain in a new direction. Chapter 3.

C2 Implementation of a functional prototype of the Dyconit consistency model in a publish-subscribe environment. Chapter 4.

C3 Evaluation of the Dyconit implementation under different circumstances. My results indicate that Dyconits can successfully manage the consistency and performance trade-off in high workload scenarios depending on the configuration. Chapter 5.

1.5 Plagiarism Declaration

I confirm that this thesis is my own work, is not copied from any other source (person, Internet, or machine) unless credited and properly cited, and has not been submitted elsewhere for assessment. I understand plagiarism is a serious academic offence that can

result in severe consequences. I acknowledge the importance of academic integrity and have made every effort to uphold it in this work.

1.6 Thesis Structure

This thesis is structured in the following manner: Chapter 2 provides the necessary background of the relevant terms and topics for this thesis. It introduces the publish-subscribe system and discusses different consistency models. Chapter 3 analyses the requirements of publish-subscribe systems and describes the design of a Dyconit system for these domains. Chapter 4 explains the details of the implementation of my prototype. Chapter 5 evaluates the performance and effectiveness of my prototype under different circumstances. Lastly, Chapter 6 summarises this thesis's main contributions, points out this work's limitations, and suggests areas for future research.

2

Background

In this chapter, I further explore the concepts relevant to my project. I delve into consistency in distributed systems, Conits, a consistency model, and I examine Dyconits, an extension of conits, capable of dynamically adjusting consistency levels dependent on the client's needs. Finally, I summarise the publish-subscribe pattern and work related to my thesis.

2.1 Consistency

Consistency with respect to data in distributed systems means that a consistent state requires all relationships between data items and replicas to agree and accurately reflect the intended state (8).

Consistency in distributed systems is often considered from two perspectives. The first is **data-centric consistency**, which focuses on maintaining a synchronized view of the data itself. Strong consistency models, such as linearizability, ensure that every read operation simultaneously reflects the most recent write operation across all nodes. In contrast, eventual consistency models allow for temporary inconsistencies between nodes but guarantee that all nodes will eventually converge to the same consistent state after all updates propagate through the system.

On the other hand, **client-centric consistency** in distributed systems focuses on enforcing consistency from the perspective of individual clients or users. Models following this pattern ensure that a client's updates or changes are immediately visible to that client, regardless of the current state of the data on other nodes. Client-centric consistency models include, for example, monotonic read consistency, which guarantees that a client never

sees older versions of data than previously and read-your-writes consistency model, which guarantees that a client always sees its updates immediately after making them.

In distributed systems, there is a well-recognized **trade-off between consistency and performance**. Ensuring strict consistency across all nodes in a distributed system can impose a significant performance overhead as each operation may involve synchronization across multiple nodes. On the other hand, relaxing consistency requirements can significantly improve performance by allowing nodes to operate independently and asynchronously. However, it can lead to anomalies such as stale or inconsistent data being read; additionally, it may require additional mechanisms to resolve conflicts and moderate divergent data states across the distributed system.

2.2 Conits

This section introduces the Conit consistency model, which was later built upon to create the Dyconit consistency model. To fully understand Dyconits, reviewing Conits and the consistency metrics they presented is essential.

Yu and Vahdat (7) developed a continuous consistency model to quantify the consistency of a system by defining three application-independent metrics*, numerical error, order error,* and *staleness error*. This allows the model to capture the spectrum between strict and eventual consistency. Instead of fixed consistency levels, it allows applications to express their specific requirements. In addition to quantifying consistency in terms of the three errors, the continuous consistency model allows the quantities to be bounded for any set of multiple instances of the same data object. Each data object represents one Conit.

The staleness error bounds the consistency in terms of real-time. It is represented by the time elapsed since the data was last synchronized. The order error evaluates the order of operations and allows for tolerance for out-of-order execution. Numerical error quantifies how much divergence from the ideal consistency level is acceptable. Each update message is assigned a weight to represent the change in the state caused by the update. Numerical error is defined as the sum of weights of the updates a Conit has not seen yet.

2.3 Dyconits

Building on the work of Conits, Donkervliet et al. introduced Dyconits (6). The following section is a summary of their work.

Their team focused on implementing Dyconits in MVEs, specifically Minecraft-like games. The gaming industry is one of the biggest industries in the world (9), and it drives the progress of large-scale distributed systems (10, 11, 12).

Dyconits represent a dynamic version of the previous Conit model. They bound inconsistency dynamically and optimistically by serving as middleware between the game simulation and the networking layer. With this approach, they were able to increase the scalability of MVEs by supporting up to 40% more concurrent players and reducing network bandwidth by up to 85%

2.3.1 Dyconits Design

To inform the design of the Dyconit model for MVEs. Five primary requirements were formulated. This subsection covers the requirements and how they influenced the design of the model.

(R1) Reduce system-wide network usage. Even servers situated within high-performance networking environments, such as data centres, constitute a scalability bottleneck. The bottleneck arises from the quadratic or even cubic increase in required bandwidth based on the number of users (13). The Dyconit model provides an improvement by introducing inconsistency. The system merges messages concerning the same state, leading to large bandwidth reductions for frequently modified states. Additionally, system-level overhead is reduced by queuing the state updates, which allows them to be sent in batches.

(R2) Quantify and optimistically bound the inconsistency. The introduced inconsistency can improve performance, but it can negatively affect the quality of experience depending on the type and extent of the inconsistency (14, 15). Each Dyconit represents an arbitrary state, an arbitrary subset of the MVE. It can bound the staleness error and the numerical error. The bounds are dynamically updated based on changing interests. The system quantifies the inconsistency and forwards state updates to the corresponding client if it has exceeded the bounds to bound the inconsistency.

(R3) Allow fine-grained control over system inconsistency. Due to different priorities for different users, the system has to allow inconsistency on a per-user basis to achieve the most benefits. For example, users are more interested in actions performed by their friends (16), or in their immediate environment (11, 17). This further correlates with the need for **(R4) allowing consistency bounds to be modified dynamically.** Due to a change in interest over time, the system must be able to adapt dynamically to prevent high inconsistency for the state that the user is currently interacting with. To benefit each user **(R5), the system has to be simple yet flexible.** To achieve these goals, the

Dyconits system manages each Dyconit dynamically and automatically. The system uses policies to dynamically update the Dyconits to match the consistency requirements of each user.

2.4 Publish-subscribe pattern

The publish-subscribe pattern is a messaging paradigm where publishers send messages without specifying recipients, and consumers receive messages without knowing the source(18). Messages are classified into topics, and the consumers express interest in specific topics, and the system ensures that relevant messages are delivered to them. This loosely coupled nature of producers and consumers enhances scalability and flexibility, making it ideal for distributed applications such as event notification, real-time analytics, and content distribution (19). Figure 2.1 shows an overview of such a system, with the producer sending messages to one or multiple message brokers and the brokers forwarding it to the consumer. In practice, the consumer can also poll for the messages themselves (20).

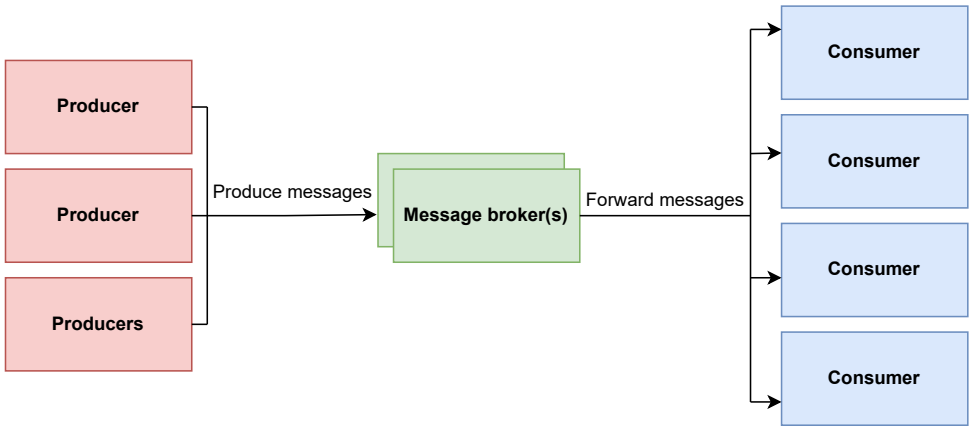

Figure 2.1: Overview of a publish-subscribe messaging pattern in distributed systems.

2.5 Related Work

This thesis is not the first to aim to implement Dyconits in a publish-subscribe environment. In 2023, Brandsen (21) introduced Hestia, a general Dyconit middleware for publish-subscribe systems. My thesis, however, takes a different approach to the implementation. Most importantly, my work preserves the loosely coupled nature of publish-subscribe platforms.

3

Design

This chapter presents the design of a Dyconit system for publish-subscribe systems, answering **(RQ1)**. The chapter starts by addressing the system requirements, followed by the definitions of consistency within a publish-subscribe system and the design itself. Afterwards, I give an overview of the system, followed by a detailed description of each of the system's features.

3.1 Functional requirements

FR1 Publish-subscribe architecture pattern support: The system should extend the original use case of Dyconits in MVEs to a publish-subscribe system.

FR2 System state consistency range based on priority: The system should support a range of consistency levels for subsets of its state based on priority levels to prioritise the dynamically determined, most relevant data.

FR3 Dynamic consistency updates based on application needs: The system should adjust the consistency bounds of an application based on its performance and the priority of relevant updates.

FR4 Configurable node settings: Initial settings should be configurable to allow users to specify the best settings for their use case. The settings should include at least the initial bounds and their dynamic behaviour.

FR5 Real-time monitoring: The system should collect data about its performance to provide insights into how the Dyconit system is influencing the behaviour of the application at runtime, including metrics for consistency, performance, and possibly other relevant data.

3.2 Non-functional

NFR1 Ensure a small overhead throughput for Dyconits communication: The goal of the Dyconit system is to improve the performance of a system; high overhead is counter-productive in this endeavour.

NFR2 Ensure high responsiveness to system state changes: The system should dynamically adapt to the current workload of the system to ensure its current consistency levels are relevant to the state of the system. The system should be designed to adjust immediately after detecting sufficiently big enough changes in workload, not accounting for network and processing delays.

3.3 Quantifying consistency in publish-subscribe systems

To extend the concept of Dyconits to publish-subscribe systems **(FR1)**, I first need to specify the consistency quantifiers in such systems.

Dyconits original measure error in two ways(6), numerical error and staleness error. For my purposes, these errors need to be redefined for the publish-subscribe environment. I need to define the data the systems need to synchronise to translate the errors to publish-subscribe systems. The state in publish-subscribe systems is determined by the messages of each topic. As such, the consistency errors will be redefined in terms of the messages in the topics each consumer is subscribed to.

Staleness error was originally defined by the time elapsed since the data was last synchronised. This is achievable in publish-subscribe platforms as well. Therefore, the staleness error will be determined by comparing the time the last message consumed per topic was produced with the current time.

Numerical error was defined as the sum of weights of the updates a Conit has not seen yet. Where the weights are assigned by the change in the state, they present for the client. Representing the change of state for each message in publish-subscribe systems would require some additional processing on the producer side. And due to the loosely coupled nature of publish-subscribe systems, it could prove troublesome to decide. Instead, the numerical error will be defined as the difference between the number of messages consumed and produced per topic.

3.4 Global and local topic priority

To enable the system to treat subsets of its state depending on its priority **(FR2)**, I introduced the concept of global and local topic priority.

The global topic priority represents a topic's importance overall compared to all other topics. Depending on the use case, it could be dynamically influenced by many things; for example, some events could be more or less important depending on the current time. In the design, the global priority should be at least dependent on the production rate of the topic because higher-producing topics will cross the numerical error bound more quickly.

The local topic priority, on the other hand, is specific to each consumer. In the current design, it depends on the configuration of the consumer and is used to enforce bounds and prioritise important events. Opposed to the global topic priority, the local priority is only determined based on the topics to which each consumer is subscribed.

3.5 Overview

Figure 3.1 depicts the design of my Dyconit system. For simplicity, the message brokers, as seen in figure 2.1, have been left out; however, they are still present in the system. The design is composed of three primary components: *Dyconit Manager, Dyconit Producer* and *Dyconit Consumer*. To implement the Dyconits model into the publish-subscribe system, the system should introduce bounded inconsistency and improve the system performance**(FR1)**. To achieve this consistency and performance trade-off in my design, I focused on the hardware resource usage of the consumers.

3.5.1 Dyconit Manager

To accommodate the ability to dynamically update consistency bounds **(FR3)**, I present the **Dyconit Manager** (**1** in figure 3.1). The Manager serves as a central consistency management unit. The centric design has multiple advantages and disadvantages. An advantage is that a central node can enforce stricter consistency guarantees across all topics and nodes. It can ensure that all nodes adhere to a specific consistency model. It facilitates the synchronisation and coordination of operations across distributed Kafka nodes. The central node can coordinate the entire system, monitor its state and enforce consistency based on a complete picture of the system. Furthermore, it also serves as a central point for monitoring the system and its performance. Among its limitations is that it presents a single point of failure. If the central node fails or becomes unreachable, it will disrupt

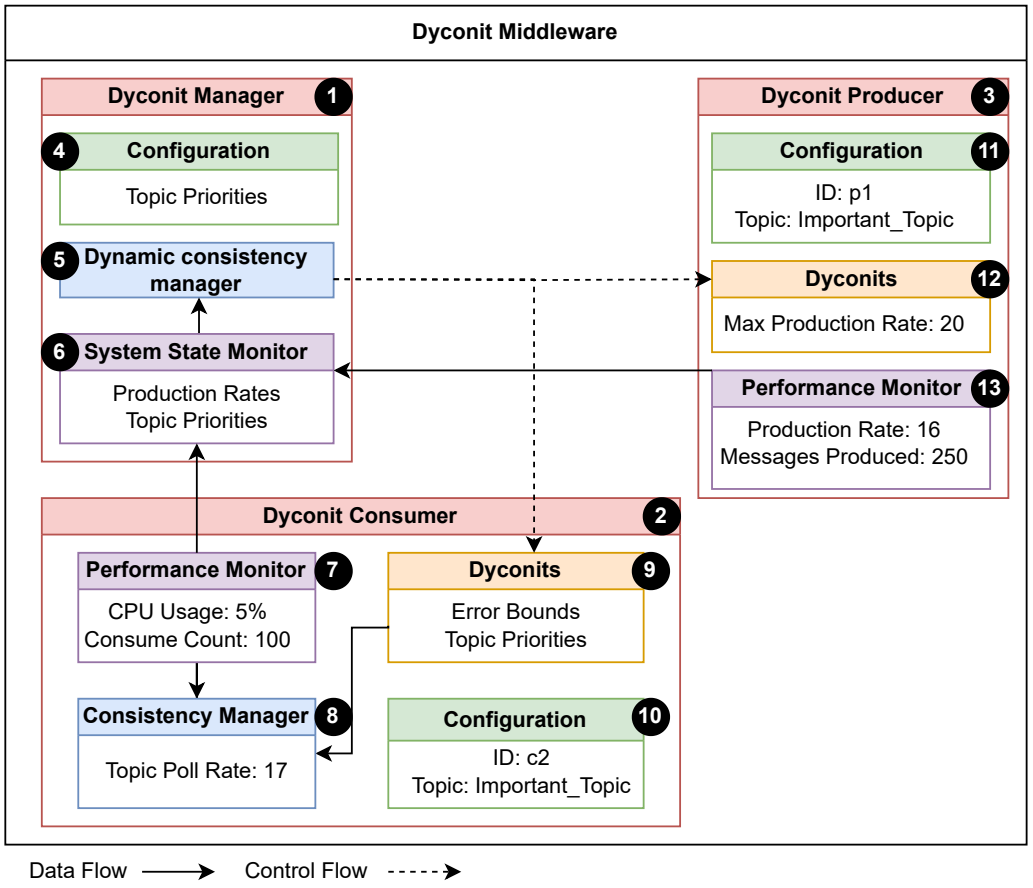

Figure 3.1: High-level overview of a Dyconit system in a publish-subscribe architecture.

the system's ability to determine new bounds dynamically. Furthermore, it may become a scalability and performance bottleneck. Especially under high throughput scenarios, all consistency-related operations must pass through the central node, potentially limiting scalability and increasing latency for data operations. The limitations and potential solutions are further discussed in section 6.2.

The Dyconit Manger consists of the following parts: **Configuration (4)**, the Manager has an associated configuration file to fulfil its part of **(FR4)**. A user is able to configure default global topic priorities and a range of settings to determine the behaviour of the dynamic consistency update generation. Different configurations act as policies furthering the adaption of Dyconits into publish-subscribe systems **(FR1)**. **System state**

monitor (6) is a collection of collections. To manage consistency dynamically, the Manager has to keep track of the distributed system's state, which consists of current global topic priorities and the performance of the nodes. Other nodes in the system periodically update these collections with their up-to-date performance and status. **Dynamic consistency manager (5)** is in charge of managing the consistency of the system. It does this by considering the current state of the system and the configuration. It dynamically updates the error bounds of consumers and global topic priorities when needed, partially fulfilling **(FR3)**.

3.5.2 Dyconit Consumer and Producer

To successfully extend Dyconits onto the publish-subscribe architecture, the consumer needs to get additional features and functionalities while maintaining its traditional consumer role. The design serves the intention to leverage the consistency and performance trade-off to implement the Dyconits model **(FR1)**. In the case of the consumer, the trade-off signifies means lowering the usage of hardware resources while bonding the consistency levels.

To achieve this, the **Dyconit Consumer (2)** consists of the following parts: **Configuration (10)**, just like the Manager, the consumer is fully configurable to support a range of consistency bounds and behaviours **(FR4)**. The most important settings include default consistency bounds and local topic priorities. **Performance monitor (7)** monitors the hardware performance of the consumer and its consistency state inside of the system, which includes the relevant information to compute the consistency errors, so the number of messages consumed per topic and the time the last message was consumed. This is further enable real-time monitoring of the system **(FR5)**. **Dyconits (9)** refers to a set of local collections related to the consistency of the system. That is the list of local topic priorities and the error bound for each topic for each error. **Consistency manager (8)** is in charge of enforcing the consistency of the system. As both the numerical and the staleness error are related to the consumer, consistency enforcement can happen locally to each consumer. The consistency manager strives to specify the usage of local hardware resources while enforcing consistency bounds.

The **Dyconit Producer (3)** requires fewer additions than the consumer. To complete **(FR4)**, it includes a **configuration (11)** file. Depending on the use case, this can streamline producer configurations by allowing easy selection of topics to be produced and strict limitations on the production rate. The producer monitors its state with a **performance monitor (13)** to help inform the consistency state of the system. This includes

the production rate and the messages produced, both per topic. To allow for dynamic consistency updates based on application needs **(FR3)**, the producer has to keep track of a dynamically adjustable maximum production rate in **Dyconits (12)**.

3.6 Dynamically adjusting consistency bounds with a central management unit

To further facilitate dynamic consistency bounds updates **(FR3)**, the consumers need to communicate with the Dyconit manager. Each consumer can individually request new bounds depending on its needs. The goal of the system is to improve performance by introducing inconsistency. Therefore, new consistency bounds are requested based on the consumer's performance. Donkervliet et al. were able to improve performance by allowing batching of state updates, which in turn reduced network bandwidth (6). Batching is, however, already present in some publish-subscribe platforms, for example, Apache Kafka (20). Therefore, there is not much space for improvement in network bandwidth, as every message must be delivered. Instead, I choose to focus on the performance in terms of hardware resource usage. Figure 3.1 mentions CPU usage; however, by design, this can be configurable so that a programmer can choose the metric that matters the most in their specific use case. The Dyconit Consumer periodically measures its performance and sends the report to the Manager. The report includes relevant information to update the overall state of the system. If the Dyconit Consumer crosses a configurable performance threshold, its next report will include a request for new bounds.

Upon such request, the Dyconit Manager responds to the consumer with the updated bounds, finally satisfying**(FR3)**. The Manager determined the new boundaries based on the global priority of each topic. The Manager linearly increases each bound by some predefined step scaled by the priority of the topic. Finally, after receiving the update, the consumer reevaluates it based on its local set of topic priorities to satisfy the need to support a range of consistency levels based on event priorities **(FR2)**.

3.7 Enforcing consistency

The inconsistency needs to be dynamically and optimistically bound to translate the concept of Dyconit to publish-subscribe systems**(FR1)**. The Dyconit Manager satisfies the dynamic part by updating the bounds at runtime when requested. However, the bounds still need to be optimistically enforced. In my design, the consistency of the system is

enforced by changing the polling rate of Dyconit Consumer. The Dyconit Consumer measures the hardware usage of a specified component, for example, CPU or GPU usage and tries to achieve its configured usage by adjusting its *polling rates* depending on the local topic priorities. While determining the polling rates, the Dyconit Consumer also considers its consistency error bounds.

If *the numerical error bound* is exceeded, the Dyconit Consumers polling rate is set to at least the production rate of the relevant topic. Therefore, the numerical error will not increase anymore. If the staleness error is exceeded, the Dyconit Consumers polling rate will increase the local topic priority. Once *the staleness error bound* is exceeded, the rate at which topic priority is increased will increase as well. This leads to an exponential increase in local topic priority. Therefore, Dyconit Consumers will quickly change its focus to the stale topic. Both error bounds are optimistic as the exceedance of the bounds causes the Dyconit Consumers to change their behaviour, leading to the error values temporally exceeding them.

In addition to the polling rate of the consumers, should the specific use case allow it, the Dyconit Manager can also *throttle the production rate* of the Dyconit Producer to help enforce consistency. Suppose a settable percentage of Dyconit Consumers are incapable of maintaining their consistency bounds and are requesting new ones. In that case, the Dyconit Manager will identify a relevant topic with the smallest priority. If this topic is sufficiently low priority, the Dyconit Manager will message every producer producing on this topic. The update includes a maximum production rate for the topic and the time this has to be obeyed. Throttling the production rates is certainly not possible in every scenario. However, when it is, it will help the system maintain a higher consistency level in a high workload period.

3.8 Comunication pattern between nodes

To fully accommodate dynamic updates **(FR3)** and ensure a responsive system **(NFR2)**, it is essential for the nodes to communicate in an efficient manner. Figure 3.2 presents an overview of the communication pattern of the system. All of the communication is happening through the existing publish-subscribe system on topics specific to each node to ensure a node never has to process updates that are not intended for it. Using the existing infrastructure maintains the loosely coupled nature of publish-subscribe systems and does not introduce any additional overhead.

Crucially, the Dyconit Manager needs to have up-to-date information regarding the performance of the system. Consumers have to report which topics they are subscribed to and whether they need new bounds. Additionally, Producers have to specify their production rate for every topic. The Manager can then calculate the overall topic production rates for all producers combined. On the other hand, the Dyconit Consumer needs to be updated with the total information regarding the numerical error and the current global topic priorities.

The Dyconit consumers and producers send reports periodically to provide an up-to-date system overview. To achieve high responsiveness, the Dyconit Manager always responds to the Consumer reports with production rates and total messages produced per topic to facilitate the computation and enforcement of the numerical error bound and topic priorities to inform polling rate selection per topic. The Dyconit Manager may also decide to throttle any producer if needed. Additionally, after receiving a producer report, the Dyconit Manager may inform the consumers of the new production rates or throttle a producer, provided they are sufficiently different. The responsive nature of the Manager, to both the producers and consumers, ensures the system behaves appropriately based on the changing conditions**(NFR2)**.

The communication related to consistency creates some overhead throughput. To improve the performance of the system, this overhead throughput should be as low as possible**(NFR1)**. In my design, the overhead in terms of additional messages the Dyconit Consumer has to process has an upper bound of:

$$\frac{number\_of\_producers \cdot producer\_report\_rate + consumer\_report\_rate}{combinded\_production\_rate} \tag{3.1}$$

The additional messages that need to be processed are responses from the Dyconit Manager to the consumer updates, and each producer update can trigger an update for the consumers, granted the production rate has changed significantly. The upper bound grows quickly; however, in systems with a stable production rate, there are few updates caused by the producer reports due to the stable production rate. Therefore, the overhead is closer to the lower bound:

$$\frac{consumer\_report\_rate}{combinded\_production\_rate} \tag{3.2}$$

These equations allow programmers to find a suitable value for the consumer and producer report rate to satisfy the low overhead requirement **(NFR1)** based on the expected behaviour of their system.

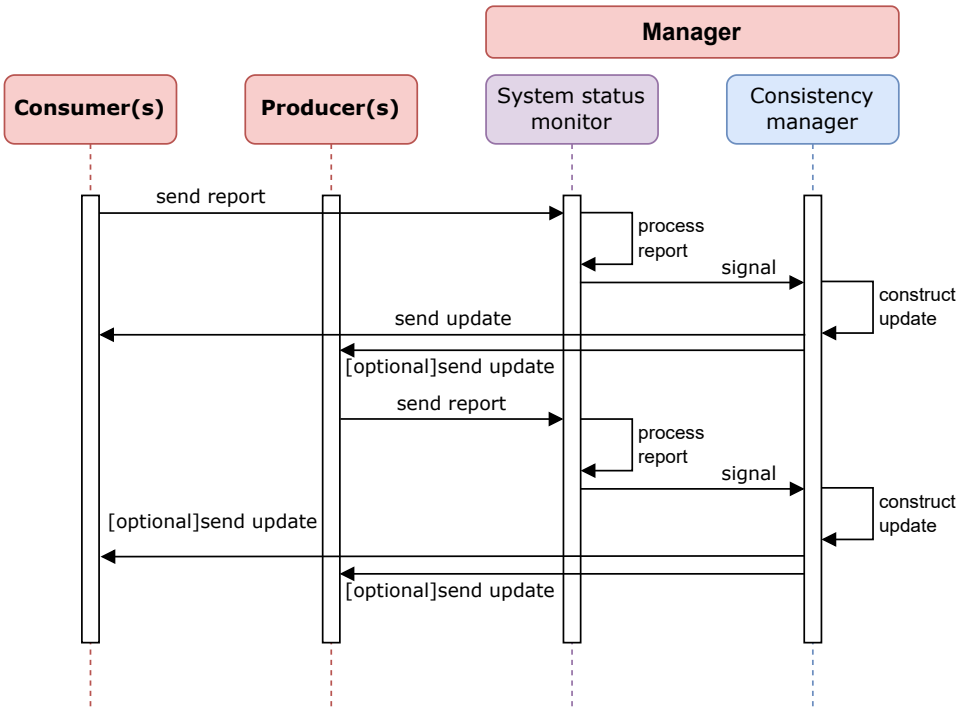

Figure 3.2: Communication pattern overview.

3.9 Summary

I reached this design through an iterative design process following the principles of At-Large (22). In this section, I have described a design for a Dyconit system integrated into a publish-subscribe environment. I have identified and satisfied the system's functional and non-functional requirements. I quantified the consistency level of a publish-subscribe environment and introduced solutions to manage it based on application needs dynamically and optimistically. The design has three main components: the Dyconit Manager, the Dyconit Consumer, and the Dyconit Producer. They communicate at run time to dynamically determine and adjust the consistency of the system based on the numerical error, staleness error and topic priorities. This section answers how to design a Dyconit system for a publish-subscribe platform **(RQ1)**.

4

Implementation

In the previous chapter, I proposed a design for integrating Dyconits within a publish-subscribe platform. The design must first be implemented to validate its usefulness and effectiveness. In this chapter, I apply the requirements from the previous chapter to create a practical implementation and explain the thought process behind the choices and trade-offs made during development, answering how to implement a Dyconit system in a publish-subscribe system **(RQ2)**. Lastly, I discuss the biggest challenges encountered while making the prototype.

4.1 Apache Kafka as the publish-subscribe platform of choice

There are multiple different publish-subscribe platforms. While my design can be integrated into any of them, this thesis focuses on Apache Kafka as its publish-subscribe platform of choice. I have chosen Apache Kafka for several reasons. Kafka excels in handling high-throughput, real-time data streams with built-in fault tolerance and scalability. This makes it ideal for experiments where maintaining data consistency and minimizing latency are critical.

Other platforms offer high performance as well. However, the biggest advantage of Kafka is its widespread use and open-source nature. Kafka offers extensive documentation and plenty of resources and tutorials, while its widespread use ensures the implementation and experiments are highly relevant, with Apache Kafka claiming to be used by more than 80% of all Fortune 100 companies (1). Furthermore, as Apache Kafka is open-source, vendor lock-in is not a risk.

The usual Kafka setup consists of nodes that act as consumers, producers, or both. I used the Confluent.Kafka library for .NET, which offers a set of abstractions to interact

with Apache Kafka. This library simplifies the creation of nodes that can interact with Apache Kafka as consumers and producers. A working Kafka system can be extended to support Dyconits.

4.2 Quantifying topic priority

As described in section 3.4, I introduced the global and local topic priority concept to enable the system to treat subsets of its state depending on its priority (FR2).

The global topic priority is handled exclusively by the Dyconit Manager. It is represented by numbers ranging from -1 to 1, -1 not included. The higher the number, the higher the priority. It also includes special values. Any priority smaller than -1 is for extremely low-priority topics that consumers can ignore the consistency bounds of. Additionally, any priority smaller or equal to -2 means that the topic has seized production and if a consumer has consumed every message on the topic, it can unsubscribe. The priorities are initialized to default values. After every Dyconit Producer performance report, the Dyconit Manager recalcutes the priorities of every topic. Equation 4.1 shows how priorities are calculated based on the production rate of the topic compared to the current maximum production rate, scaled to be on the scale -1 to 1 and then averaged with their default priority.

$$topic\_priority = \frac{\frac{topic\_production\_rate}{maximum\_production\_rate} \cdot 2 - 1 + default\_topic\_priority}{2} \tag{4.1}$$

The local topic priority is handled by individual Dyconit Consumers. The priority range is still -1 to 1. Less than -2 means that the topic is no longer producing, and the consumer has received every message. Less than -1 means that the error bounds can be ignored. Additionally, Dyconit Consumers uses the topic priority to enforce the staleness error. Therefore, priority for stale topics can exceed 1. Each Consumer receives the global priorities from the Dyconit Manager. Upon receiving them, the consumer localises them according to its own preferences by first averaging them with its local default values; afterwards, the priorities are normalized so that the highest priority topic has priority 1 to ensure correct behaviour in cases where the consumer is only consuming one topic.

4.3 Asynchronous node design

My implementation consists of three node types: a Dyconit Manager, Dyconit Producers and Dyconit Consumers. The nodes share the same structure; they are designed to run

continuously in the background and perform various tasks at various time intervals. As some of the tasks could be required at the same time, such as polling two different topics for a new message, some kind of parallelism is necessary. I decided to use the .NET `Task` class to accommodate asynchronous programming.

Each `Task` represents an asynchronous operation. It is a higher-level abstraction allowing for the use of `async` and `await`. This simplifies the process of writing asynchronous code and makes it less error-prone than other means of achieving concurrency, like threads. It is implemented using a thread pool, which makes it efficient and means that threads are never blocked by a `Task` scheduled to happen at some time, as is often the case in this implementation, leading to overall better performance over threads.

The Dyconit Manager has two `Task`s running continuously to consume consumer and producer updates. To ensure the updates can be processed immediately **(NFR2)**, the manager creates a new asynchronous task to process each update and respond to it. This approach is highly scalable with the number of available threads, as the only point of contention is polling for new messages.

The Dyconit Consumer has a continuous `Task` for each topic it is subscribed to for consuming messages. additionally, there is a periodic `Task` running determined by the report rate, which calculates the performance of the system and generates a report. Receiving updates and calculating performance also triggers a task to determine the polling rate to ensure the polling rate is up to date with the state of the consumer and the system. Determining it asynchronously ensures it does not interfere with receiving messages or reporting performance.

The Dyconit Producer has a continuous `Task` for producing messages and two periodic `Task`s. One serves for reporting performance, just like the Dyconit Consumer. The other one determines the production rate to simulate different scenarios. Additionally, the writes for logging information for data collection and monitoring **(FR5)** are asynchronous for all nodes to prevent disturbances caused by slow writing operations.

4.4 Structure of node communication

As highlighted in the design section 3.8, the nodes in the implementation need to communicate. In the prototype, the Dyconits effectively introduce two communication channels at any point in time. Firstly, the producers and consumers report to the manager. Secondly, the manager updates the consumers and producers with instructions to enforce consistency.

4.4.1 Topic selection

To facilitate communication over the existing Kafka infrastructure, the nodes need to agree on which *topics* are used for communication. This subsection describes the topic selection and creation process for each node.

The Producers and Consumers are each assigned a configurable ID, and they create their own topic for consuming consistency updates. This is done by taking their ID and appending a corresponding suffix for consumers and producers, `_consumer_update` or `_producer_update`, respectively. Creating a topic is a simple one-time operation for confluent kafka, and one cluster can support hundreds of thousands of topics (23); therefore, the overhead caused by topic creation is not significant. Alternatively, all nodes could listen to the same topic, and the update message would indicate which node it is for. However, this approach would generate more overhead for each node, as each update message would have to be processed by every node in the system. Each report includes the ID of the node that sent it, enabling the Dyconit Manager to produce messages on the correct topic. With the ID, the Dyconit Manager can reconstruct the correct topic and produce consistency updates and the correct topics.

As my design only includes one **Dyconit Manager**, each node can send its reports on the same topic, `consumer_performance` and `producer_performance`, for consumers and producers, respectively. As the number of nodes a Manager can manage is finite, dependent on the amount of resources available, one way to extend this prototype would be to introduce multiple Managers, each with their own report topics and a topic for communication between the Managers to enable the coordination of larger distributed systems.

4.4.2 Required Information

To enable the functionalities of the system, each report and update has a standardized format and includes the information necessary for each node to function. This subsection describes the format of each message type related to consistency management.

To allow the Dyconit Manager to monitor the system, the **producer reports** include the ID, the number of messages since the last report for each topic, and an optional token to signal that production on that topic has ended. The Dyconit Manager is also interested in the production rate; however, Kafka messages, by default, include a timestamp so the production rate can be calculated. The reports also include an optional token to indicate

the producer has stopped producing that topic. The reports are structured in the following format:

```
{ID} [For each produced topic] {Number of messages produced since the last report} {Topic} [optional] q
```

The consumer reports include the ID, the CPU usage, and the total number of messages consumed for each topic. The report will include an optional token if the consumer needs new bounds. This information is used to monitor the system's state, determine how many consumers need new bounds, and check whether the consumer has consumed every message on the topic. The reports are structured in the following format:

```
{ID} {CPU usage}[For each consumed topic] {Topic} {messages consumed in topic} [optional] nb
```

The Manager's consistency updates are structured in a similar way to the performance reports. **Updates for producers** include the ID to confirm the recipient, the minimum delay between sending messages, which directly translates to the production rate, and the time the minimum delay should be in effect, called throttle time. These updates are only sent if the Dyconit Manager is allowed to throttle the production speed of topics. The update is structured as follows:

```
{ID} md {minimum delay} tt {throttle time}
```

Updates for consumers follow the same overall structure but include multiple more parameters. They have the ID, then per topic, the production delay of each topic, updated global topic priorities and total messages produced per topic. The Dyconit Consumer uses this information to enforce and calculate consistency. Had the consumer previously requested new bounds, the message would have included changes to numerical and stale error bounds per topic. The production delay is the time between messages in the topic. The consumer updates are structured as follows:

```
{ID} [each per topic consumed] mp {max topic poll delay} {topic} tp {topic priority} {topic} ne {topic messages produced} {topic} [optional] bn {change in topic numerical error bound} {topic} [optional] bs {change in topic staleness error bound} {topic}
```

4.4.3 Communication control flow

The communication follows the figure 3.2 shown in the design chapter 3. The consumers and producers periodically send their reports based on the configured report rates. Algorithm 1 shows the logic used to determine *which producers receive an update* and the consistency update for a consumer. This algorithm is called in two scenarios. It is called after every received consumer update to guarantee it regularly gets updated consistency information. Additionally, it is called after a producer reports if the production rate has sufficiently changed to notify the consumer of the change in production rate.

```
Result: Sent consistency updates for a consumer and producers.
if needs to throttle and is allowed to throttle then
    Find a suitable topic;
    Throttle topic's producers;
    Reset new bounds requests;
end
foreach topic in consumer_topics do
    if consumer consumed everyting and topic is not producing then
        Topic priority = -2;
    else
        if topic has a very low priority and new bounds requested then
            Topic priority = -1;
        else
            Topic priority = global topic priority;
        end
    end
    Construct a consumer update based on topic priorities;
    Send consumer update;
end
```

Algorithm 1: Generate consistency updates.

The algorithm uses a consumer as an argument. It determines whether to throttle producers based on the number of consumers requesting new bounds and then constructs and sends the consumer update. Therefore, after a consumer report, the consumer receives a response. Any number of producers may be updated, and after a producer report, any number of consumers and producers may be updated. This communication flow satisfies the high responsiveness requirement **(NFR2)** and allows the nodes to dynamically control consistency based on the immediate state of each node **(FR1, FR2, FR3)**.

4.5 Dynamically adjusting system performance at run time

At runtime, the behaviour of the system changes based on consistency and performance **(FR3)**. The adjustments are primarily relevant to the consumer. **The Dyconit Consumer** aims to *minimise CPU usage* while *maintaining consistency bounds*. Periodically, just before reporting performance, the Dyconit Consumer calculates its CPU usage as the processor time divided by the elapsed time. If the CPU usage exceeds a threshold the next report will include a request for new bounds. To control the performance and consistency of the system, the Dyconit Consumer changes its *polling rate* to increase or decrease the speed of consuming new events. The polling rate is determined by the calculated CPU usage, error bounds and state information received from the Dyconit Manager. Algorithm 2 shows the function `DeterminePollRate()` that decides the polling rate of each topic. The function is called after calculating performance and after receiving an update from the manager. The polling rate is determined by setting a minimum delay between consuming messages, which I refer to as the polling delay.

Result: Updated topic polling delays.
Determine the update step depending on the CPU usage and CPU usage target;
foreach *topic* in *subscribed_ topics* **do**
  if *Topic_ priority <= -1* **then**
    Set topic polling delay to maximum polling delay;
  end
  else if `IsStaleErrorExceeded(`*topic*`)` **then**
    Change topic polling delay by the update step scaled by stale topic priority;
    Increment stale topic priority;
  end
  else if `IsNumErrorExceeded(`*topic*`)` **then**
    Change topic delay by the update step;
    Set topic polling delay to at least the topic production rate;
  end
  else
    Change topic delay by the update step;
  end
end

Algorithm 2: Update polling rates.

Firstly, based on the current *CPU usage* and the *target CPU usage*, the algorithm 2 determines the update step, indicating whether the polling delay should decrease or increase

to *increase or decrease workload* respectively. The update step is based on the difference in target and current CPU usage to determine whether the polling delays should increase or decrease. The size of the step is configurable; higher values change polling rates faster, leading to potential faster adjustments to the system but increased variance in polling rates. Afterwards, each topic is individually considered. If the topic's priority is equal to -1, it is considered disabled, and the polling delay is set to the highest allowed value. Otherwise, the error bounds are checked. The polling delay is adjusted to bound the error's value if either error bound is exceeded. Information needed to calculate either error and bound it is included in every manager update.

$$update\_step = \frac{(cpu\_usage - target\_cpu\_usage)}{|cpu\_usage - target\_cpu\_usage|} * poll\_rate\_step \tag{4.2}$$

The numerical error bound is enforced by setting the polling delay so that the polling rate matches or exceeds the produce rate of the topic specified in the latest manager update. The manager already translates the production rates to delays between messages, so the Dyconit Consumer can directly use the information provided by the manager. The polling delay is first updated by the previously determined `update_step` and an additional fraction of the `poll_delay_step` determined by the `topic_priority`. Afterwards, it is set to equal, at most, the production delay between messages. The following equations bound numerical error:

```
topic_poll_delay += update_step - poll_delay_step * topic_priority
topic_poll_delay = Min(topic_poll_delay, topic_production_delay)
```

Every time **staleness error bound** is exceeded, the priority for that topic is increased, and the increment is also increased. The increment resets when the new bounds are received. This leads to an exponential decrease in the topic polling delay. Therefore, the staleness error is optimistically bound, assuming that the consumer can process messages at least as fast as they are being produced. The polling delay is first normally updated by the previously determined `update_step` and the `poll_delay_step` scaled by the `stale_topic_priority`. Afterwards, it is set to equal, at most, the production delay between messages.The following equations bound staleness error:

```
topic_poll_delay += update_step - poll_delay_step * stale_topic_priority
stale_topic_priority += stale_topic_priority_increament
```

If no errors are exceeded and the topic has a priority higher than -1, the polling delay is determined by the `update_step` and `poll_delay_step` scaled by `topic_priority`. The following equation achieves changes in polling delay based on the current performance and the importance of the topic.

```
topic_poll_delay += update_step - poll_delay_step * topic_priority
```

If configured to do so, the consistency of the system can also be enforced by *throttling* the **Dyconit Producer**. This is implemented by introducing a minimal delay between producing messages. The delay is zero until the Dyconit Manager specifies otherwise. The manager also specifies the throttle time; once the producer has been throttled for the time, the minimum delay is set to 0 again.

4.6 Determining Consistency of the system

As the manager receives the reports, the data needs to be used to determine the system's state. Five main attributes need to be determined.

Firstly, from the producer reports, the manager infers the *production rate per topic*. To properly calculate this, the manager saves the time stamp of the producer update message to get the current and next time intervals. Secondly, the manager also keeps track of the total number of messages produced per topic. Thirdly, the manager updates the *global priority for each topic* by comparing the production rates.

Additionally, the consumer reports inform the manager of *which and how many consumers need new bounds* are struggling to keep up with the current workload. Lastly, the consumers also report *how many messages they have consumed per topic*; from this, the Manager can infer which consumer has consumed all messages in a topic that has seized production.

Based on this information, the Dyconit Manager can *generate appropriate updates* in algorithm 1. The most important update the manager needs to calculate is the consistency bounds changes. The production delay is simply the inverted production rate, and the system status, such as the total number of messages consumed, directly implies other data in the updates. The consistency bounds change are calculated for each topic by scaling a configurable increment by the topic priority, as follows:

```
scale_factor = (1 - (1 + topic_priority) / 2)
num_err_bound = num_err_inc + num_err_inc * scale_factor
stale_err_bound = stale_err_inc + stale_err_inc * scale_factor
```

4.7 Configurations

One of my requirements for the design is to make the system configurable **(FR4)**. Configurable settings enable programmers to adapt my prototype for a wider range of applications. Each node has its own associated `appsettings.json` file to accommodate this. JSON was chosen due to its readability and popularity. Parsing JSON files is also very well optimized and the goto option for many configuration files. As with every node, the manager can be configured in its appsettings.json file. The configurations are split into three main parts: general node settings, initialisation of collection, and Dyconit-related information.

The Dyconit Manager's *general settings* signify the permission of the manager. Specifically, whether the manager is allowed to throttle producers, disable topics by setting priority to -1, and tell consumers to unsubscribe by setting the priority to -2. The *collections* consist of topics produced by the system, default topic priorities and the topics the manager should subscribe to. Lastly, the *Dyconits settings* are the priority threshold to determine whether a topic can be throttled or disabled, numerical error increase and staleness error increase to calculate bounds updates, the threshold to determine whether to notify consumers to a change in production rate, the threshold to determine if enough consumers need new bounds and throttle a topic and finally the throttle time and amount.

The Dyconit Consumsers's *general settings* consist of ID, default polling delay, minimum polling delay, and polling rate step, which is used when calculating new polling delays and the update step, maximum polling delay and report frequency. The *Dyconits settings* consist of the target CPU usage, CPU usage threshold to determine the need for new bounds, default error bounds and the rate at which stale error priority increases. The *collections* section includes topics to subscribe to, priority offset for those topics, and default stale priority values for each topic.

The Dyconit Producer's *general settings* include the ID, topic to produce, the time it should run for and its report rate. The *Dyconits settings* consist of settings used to simulate different workloads in the next chapter 5 the maximum and minimum delay between messages and the rate at which they change. There is no collections section.

4.8 Implemntation Challanges

I used the Confluent.Kafka library to integrate Dyconits with Apache Kafka. However, even though Kafka and Confluent Kafka are very well documented, **Apache Kafka is**

still a very complex system and as such, it was challenging to set it up. Furthermore, Kafka does not support the calculation of numerical or staleness errors or topic priorities so I had to implement a lot of additional features on top of the Confluent.Kafka library.

My design includes a lot of **asynchronous tasks**. Concurrency in programming is famously difficult. I had to determine the best times to determine new polling delays, when to calculate CPU usage and when to change production rates to simulate workloads. Furthermore, the communication pattern is also fully asynchronous so Finding approaches to ensure the tasks ran at the same time and efficiently was a big challenge.

4.9 Summary

In this chapter, I implemented the design for integrating Dyconits within a publish-subscribe platform **(RQ2)**, specifically focusing on Apache Kafka due to its high performance and scalability. The implementation involves creating Dyconit Manager, Producer, and Consumer nodes, each operating asynchronously to handle tasks efficiently. I detailed the process of quantifying topic priorities, structuring node communication, and dynamically adjusting system performance to maintain consistency and minimize CPU usage. The chapter also covered the necessary configurations for each node and the challenges faced, such as the complexity of setting up Kafka and managing concurrency in asynchronous tasks.

5

Evaluation

In this chapter, I present the experimental design and evaluation of my implementation of Dyconits into Apache Kafka and answer how to evaluate a Dyconit system for publish-subscribe systems **(RQ3)**. This includes my experiments' main findings and implications, the experiment environment descriptions, the metrics used for evaluation and how they were collected. Furthermore, I will go into detail about the choice of workload and configuration options, such as error bounds. Lastly, I discuss the limitations of my experiments.

5.1 Main findings

MF1: Dyconits can dynamically adjust performance to limit CPU usage while maintaining consistency bounds on high-priority topics, given there are also low-priority topics.

MF2: Dyconits can selectively control the consistency level of different topics and prioritise certain topics over others.

MF3: Under the current implementation, Dyconits affect the behaviour the most when the topics in the system have varying priorities and the workload is high.

MF4: Under high workload, Dyconits can reduce inconsistency by about 50% for high-priority events while increasing inconsistency for low-priority events by 10 to 200 times, effectively bounding inconsistency for high-priority events as a trade-off.

MF5: Under high workload, with topic priorities, the average CPU usage is decreased by approximately 30%, and the peak CPU usage is reduced by 25%.

MF6: Under high workload, without topic priorities, the average CPU usage is decreased by 15%, and the peak CPU usage is reduced by around 20%.

MF7: Under high workload, the current Dyconit implementation reduces throughput by around 10%, regardless of a priority difference or lack thereof.

MF8: Under low workload, the throughput, CPU usage and consistency are all largely unaffected by Dyconits.

My Dyconits model implementation leverages the performance and consistency trade-off in a publish-subscribe setting. The most interesting results were found in high workload scenarios by allowing for a bound level of inconsistency while lowering the CPU usage, slightly reducing the throughput and a minor message throughput overhead. If multiple topics with different priorities are being produced, Dyconits can improve performance by temporarily allowing lower consistency levels in low-priority topics.

This has several real-life implications; firstly, Dyconits *reduces the risk of system overload* by lowering CPU usage spikes, which prevents crashes and degraded performance, especially in high workload spikes scenarios. Secondly, the reduced average CPU load may indicate *better scalability*. However, the decrease in throughput indicates that there are still complications to solve. More research and testing could provide essential results for understanding and managing the throughput of publish-subscribe systems with Dyconits. Furthermore, the reduced CPU usage spikes and bound consistency levels of high-priority topics indicate better load management. The Dyconit system can handle spikes in data more effectively, maintaining consistent performance under varying workloads and focusing on high-priority events. The main finding and their implications together answer how to evaluate a Dyconit system for publish-subscribe systems **(RQ3)**.

5.2 Metrics

The appropriate metrics need to be chosen and evaluated to evaluate the system's implementation. The project's goal is to optimise the consistency performance trade-off. As such, the metrics were chosen best to describe the system's performance change and consistency levels. To evaluate performance, I focused on message throughput and CPU usage. To better understand changes in throughput, I also looked at overhead throughput caused by the additional messages associated with my Dyconit implementation. To evaluate consistency levels, I assessed the values of the numerical error and the staleness error. To understand the system's behaviour, I considered the arithmetic mean and maximum values of the metrics.

CPU usage and throughput indicate how efficiently the system uses the resources. A higher throughput means the system can process more messages in a given time, while a lower CPU usage means it can process messages more efficiently. Numerical and staleness errors indicate how consistent the system is at any given time. Lower maximum error values mean the consistency was more strictly bound.

5.2.1 Data Collection

The prototype logs the relevant information during runtime to collect the necessary data. For the Dyconit consumer, the logged information consists of the number of messages consumed and the CPU usage. Other consumer-relevant metrics can be calculated based on the logged data and logging frequency. The Dyconit producer periodically logs the number of messages produced, directly implying the production rate.

The inconsistency quantifiers are also recalculated from the logged data. Logging numerical and staleness errors on the consumer would be vulnerable to inaccuracies as the consumers do not always have up-to-date data and are affected by delays caused by the Dyconit manager. However, calculating them from the number of messages consumed and produced always gives an accurate consistency measurement with precision based on the logging rate.

5.3 Experiment Deployment

The experiments were run on an ASUS TUF Gaming A15 FA506QM laptop. The CPU was an AMD Ryzen 7 5800H with a clock speed of 3201 Mhz and 32 GB of DDR4 RAM.

5.3.1 Docker

A choice of environment is necessary to run the implementation. Ideally, the system would run on a real distributed system consisting of multiple nodes. Due to time and project scope, this was not practically achievable, so instead, the environment is simulated on a single computer.

Each node will be run in its own Docker Container to achieve results closer to a real-life scenario. Using Docker to run the different Kafka nodes, in this case, the producers, consumers and the manager, present several advantages that enhance the validity of the implementation and the experiments run on it.

Firstly, Docker containers ensure a certain isolation level, as each container operates in its environment. Thereby eliminating the potential for conflicts between dependencies or

configurations. Furthermore, the isolation provides consistency across experimental runs and environments, enabling reproducibility and simplifying debugging. The consistency of Docker images guarantees the code will run identically regardless of the underlying system. This ensures the implementation and further experiments are reproducible and verifiable across different setups.

5.4 Experiment Configuration

As highlighted in the previous chapters, my implementation has many configurable variables. The values were fixed for each experiment to obtain exact and reliable results. The most significant variables are the predefined global topic priorities, consistency error bounds, CPU usage thresholds for the consumer, and production delay to control the producers' production rate. Additionally, the production rate directly affects the experiment workload. The report rate impacts the system's responsiveness as it represents how often the producers and customers update the manager.

5.4.1 Workload design

The experiments' workloads should represent real-life scenarios as closely as possible. However, the environment also limits the workload, as setting it too high could make it vulnerable to systematic errors stemming from overall system strain instead of the Dyconit implementation.

Real-life Kafka applications can reach billions of messages a day, so multiple tens of thousands of messages per second, and it can peak at over a trillion messages per day (24). As such, two workloads were designed: one for peak workloads and one for average workloads. By testing, it was determined that the system could handle at most three producers producing around 150 to 175 messages a second before the performance of the system became unstable. This was chosen as a high-workload simulation. To further increase the workload and simulate high-priority and high-cost events, an otherwise unimportant calculation is repeated ten times for high-priority messages and once for other messages. This ensures a higher peak workload. Testing found a low workload of around 30 messages per second per producer, which is a workload when a standard Kafka consumer could reliably keep up with three topics. The delay between each message is randomised in a small range to reflect a real-life scenario better.

5.4.2 Variable configuration

There are no rules for finding the best variable setting for every scenario. As such, most variables were found using preliminary testing and settling for values that produced the most promising results. Importantly, this does not diminish the relevance of the results, as, in a real-life scenario, you could also search for the best values by testing.

For topic priority, the values -0.5, 0, and 0.5 for low, medium, and high-priority topics worked well to produce different consistency bounds for each. Any three equal numbers will suffice for measurements with equal topic priorities, so I used 0.1. The default consistency bounds for high workloads were 1000 messages and 10 seconds for the numerical and staleness errors, respectively. For a low workload, the default values were 10 messages and 1 second. However, the bounds are dynamically updated at runtime based on priority and production rate. The reported frequency was set to 5 seconds during every experiment as it was found to be sufficiently flexible while minimalising overhead.

5.5 Experiments

The goal of the experiments was to determine the effect of my Dyconit implementation under different workloads and observe how topic priority affects the system's behaviour. Three experiments were conducted based on the workloads and variable configurations, and each measurement was repeated ten times to ensure statistical significance. Additionally, each experiment consisted of 3 producers running for 2 minutes with the specified workload and four consumers, 2 Dyconit consumers and two standard Kafka consumers. Table 5.1 summarises all the experiments.

E1 The first experiment represents the average workload. It has a low production rate, averaging roughly 28 messages per second, and topic priorities of -0.5 and 0, 0.5.

E2 The second experiment represents a high workload, with a high production rate of, on average, 165 messages per second. The topic priorities for this experiment are all equal.

E3 To determine the effect of topic priorities, the third experiment consists of a high workload and topic priorities of -0.5 and 0, 0.5. This experiment represents a peak workload where the production rate is significantly higher than the achievable consumption rate.

Note that there is a clear space for a fourth experiment with equal priorities and a low workload. However, in a stable system with equal priorities, Dyconits will not introduce any difference in behaviour; thus, the results of Dyconit consumers will be the same as those of standard consumers. Preliminary testing confirms this suspicion.

Experiment	Workload	Topic priorities	Workload Runtime [minutes]
1	Low	High, Medium, Low	2
2	High	Equal	2
3	High	High, Medium, Low	2

Table 5.1: Experiment setups

5.6 Experiment Results

This section presents the results of the prototype experiment. I conducted several experiments to evaluate the prototype's performance under different circumstances and used several metrics to determine the system's performance and consistency.

5.6.1 Consistency levels

The box plots in this subsection depict the numerical and staleness errors across the three experiments. I measured the average and maximum values for each implementation and topic. In the box plot, each topic has a separate colour. A box of light shades of each colour represents the averages, while a box of dark colours represents the maximum values. The graph shows the results for each topic, first for the Dyconit implementation and then for the standard implementation. Topics are sorted by priority, from the highest priority on the left to the lowest on the right. Each graph represents the values measured across 10 experiments.

The figures 5.1 and 5.2 show inconsistency in **E1**. All the errors are very low, except for two outliers in each figure on the Dyconit side. This result is expected as in a low workload scenario, the system should not introduce inconsistency. Furthermore, the errors are within the default configured bounds. These two figures clearly show that under low workload, there is no meaningful difference in the consistency levels of a consumer with Dyconits compared to a consumer without.

The average numerical error for both implementations is 0, and the max is 1. Apart from the low-priority topic, in the Dyconit prototype, the numerical error sometimes temporarily reaches more than 10 messages but always decreases to 0 or 1 by the next update. The staleness error follows the same trend as the numerical error. Both implementations averaged 0 seconds of staleness error; however, a few outlier runs of the Dyconit implementation reached around 5.5 seconds.

The figures 5.3 and 5.4 show inconsistency in **E2**. The most apparent is the fact that the areas of the boxes overlap. This makes sense, considering the topics all have the same priority and thus the same expected error bounds. The Dyconit implementation introduced some inconsistency; this is expected as **E2** represents a high workload scenario. The maximum numerical error of the Dyconit consumer compared to the Kafka consumer was between 61.50% to 101.41% higher, while the average increased by 74% to 116%. The maximum increased from around 4400 messages to about 7200 to 9000 messages. The average increased from 2000 messages to 3500 to 4500 messages.

The staleness error followed a very similar trend. Its maximum increased by 51% to 86%, while its average increased by 54% to 87%. The maximum increased from around 30 seconds to about 47 to 50 seconds. The average increased from 16 seconds to about 24 to 30 seconds.

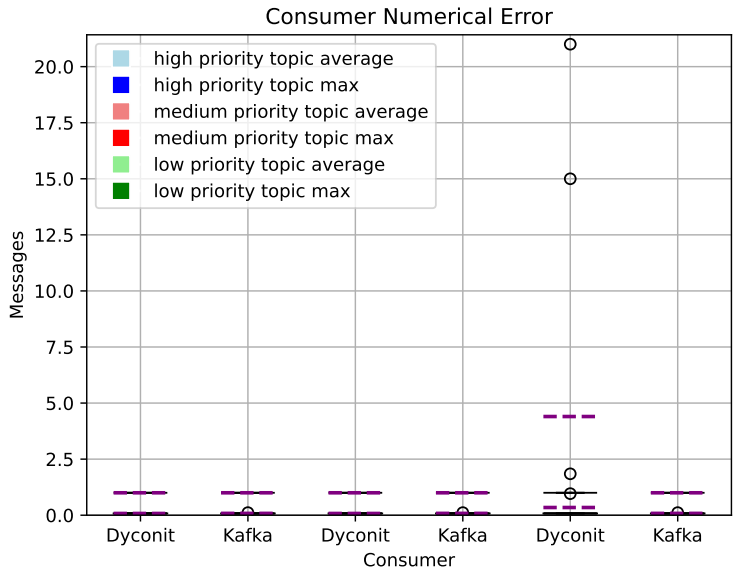

Figure 5.1: The effect of Dyconits in low workload scenario with topic priorities on numerical error in E1.

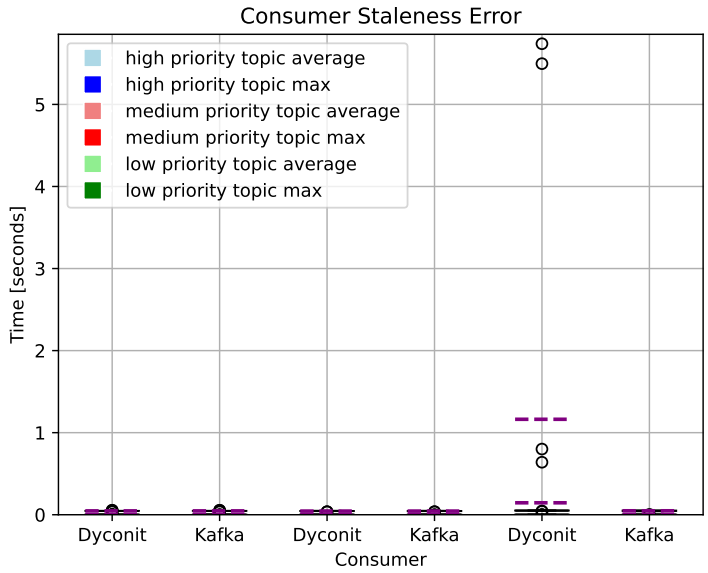

Figure 5.2: The effect of Dyconits in low workload scenario with topic priorities on staleness error in E1.

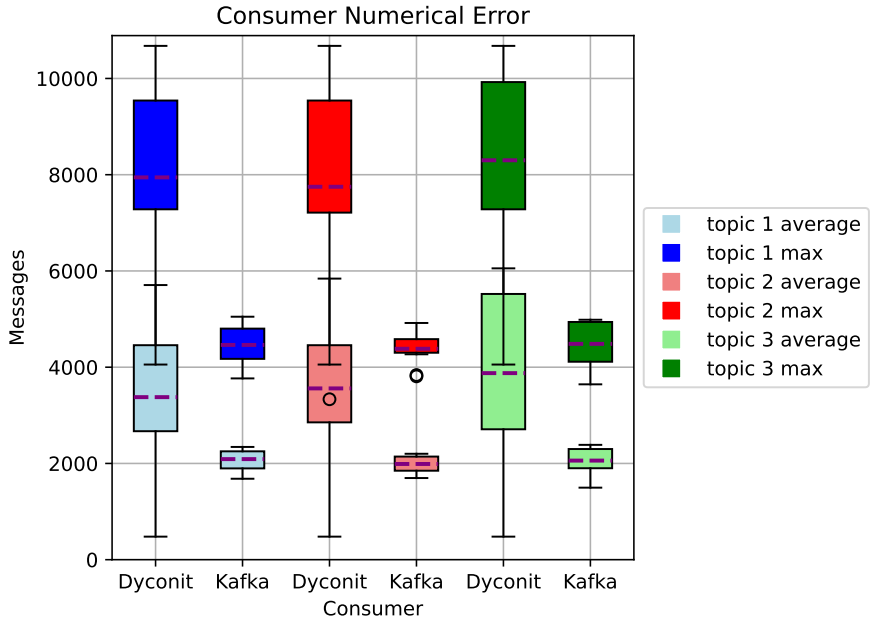

Figure 5.3: The effect of Dyconits in high workload scenario without topic priorities on the numerical error in E2.

The results are expected; my prototype introduced equal inconsistency to every topic. Section 5.6.2 considers the effect of the inconsistency on the system's performance.

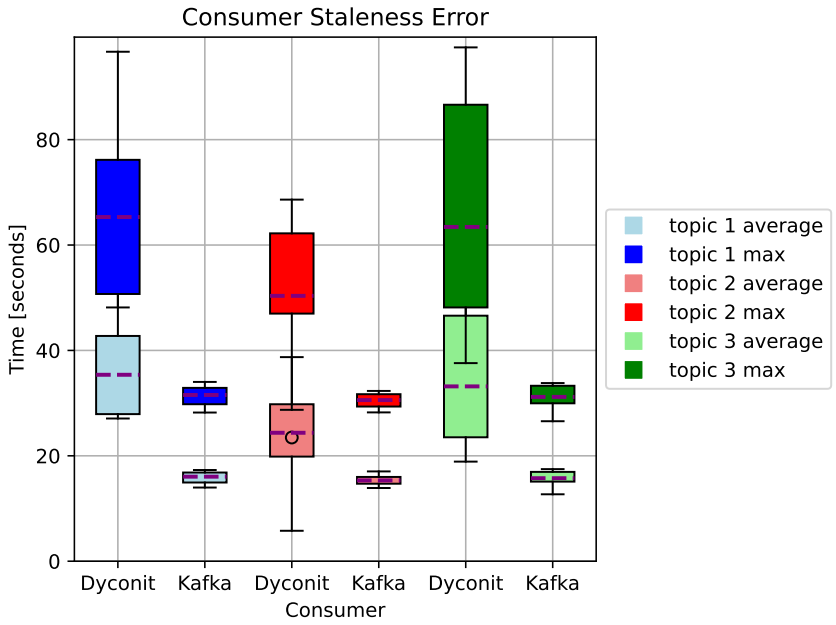

Figure 5.4: The effect of Dyconits in high workload scenario without topic priorities on staleness error in E2.

Figure 5.5 illustrates the impact of Dyconits on consumer consistency for all topics in terms of the numerical error in **E3**. This experiment is the best example of the Dyconit functionality. The consistency level of the low-priority topics decreased while the consistency level of the high-priority topics increased. The consistency of the medium-priority topic also decreases, but not as much as the consistency of the low-priority topic. My implementation decreased the maximum numerical error for the high-priority topic from 9000 to 4300 messages; the average decreased from 4000 to 1800 messages. On the other hand, the maximum for medium- and low-priority topics increased from 250 to 4200 and 17500 messages, respectively. The average increased from 35 to 870 and 8500 messages.

For high priority, my implementation has a 51.11% lower maximum numerical error. At the same time, the medium and low priority topics have a 1730% and 6046% increase, respectively, in maximum numerical error. The averages follow a similar trend. High-priority numerical error is decreased by 52.28%, while medium and low are increased by 3295% and 19038%, respectively.

Figure 5.6 shows the staleness error in **E3**. The graph is similar to numerical error, and the data follows the same trends. Dyconits decreased the maximum staleness error for the high-priority topic from 60 to 30 seconds and the average from 30 to 15 seconds. However,

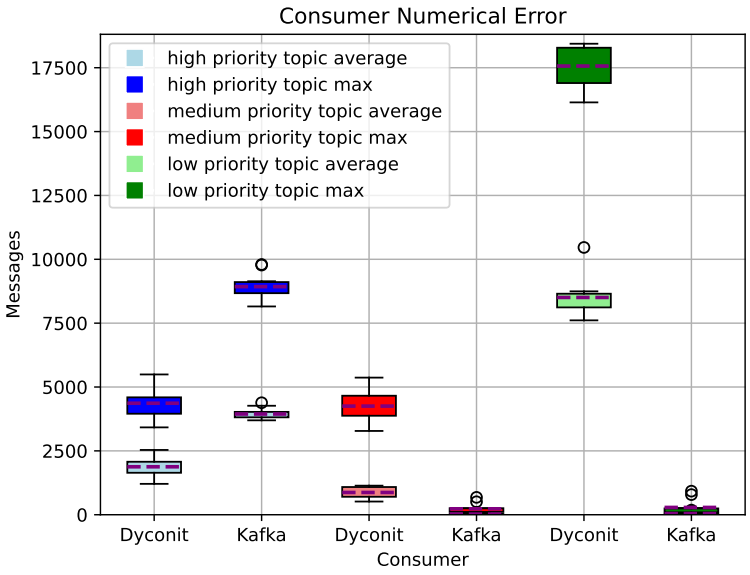

Figure 5.5: The effect of Dyconits in high workload scenario with topic priorities on numerical error in E3.

the maximum stainless error for the medium- and low-priority topics increased from 6 to 30 and 109 seconds, and the average increased from 5 to 10 and 55 seconds, respectively.

My implementation lowers the maximum staleness error by 46.51% and the average by 43.27% for the high-priority topic. The values in the same order for the medium- and low-priority topics are 505%, 205%, 1665% and 1122%. Figures 5.5 and 5.6 demonstrate that Dyconits can selectively affect performance in high-workload scenarios.

The results follow the expected trend. Dyconits increase the consistency of high-priority topics at the cost of decreasing the consistency of low-priority topics. However, the decrease in consistency is rather high. Future work should focus on more experiments and research to grasp the full extent of this trade-off.

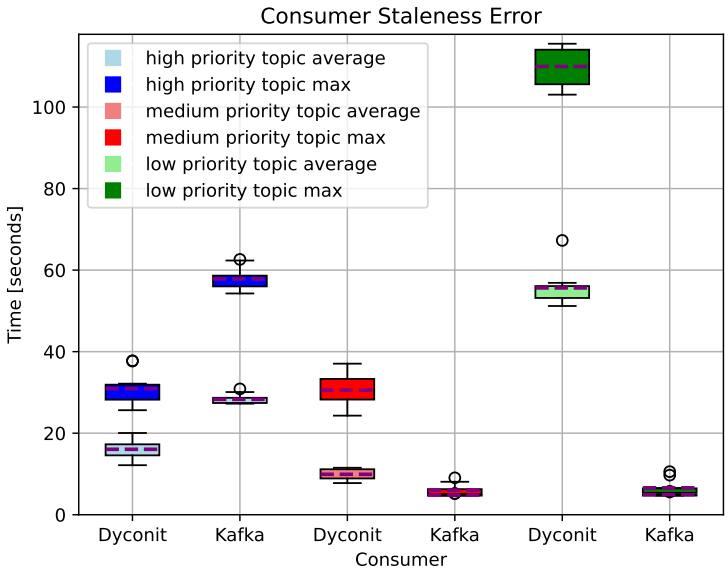

Figure 5.6: The effect of Dyconits in high workload scenario with topic priorities on staleness error in E3.

5.6.2 Performance

The following plots depict the performance of the system in the experiments. The box plots show my implementation on the left compared to a standard Kafka consumer on the right. CPU usage plots include maximum usages and the mean usages plotted on a box plot after 10 measurements. In contrast, the throughput plots only include the mean average among the 10 runs of each experiment. The maximum throughput is left out as it was generally the same for every measurement. They each depict either the CPU usage or the throughput error in one experiment.

Figures 5.7 and 5.8 show the performance of the system during **E1**. Figure 5.7 shows that in **E1**, the CPU usage is almost the same for both the Dyconit consumer and the standard consumer. Figure 5.12 shows that the throughput is also the same. This makes sense, as E1 was supposed to represent a workload a consumer can comfortably handle within its performance and consistency bounds. The changes in throughput are within half a per cent, and the CPU usage is 3% lower on average.

Comparing the performance and consistency results for **E1**. The system allowed for a minimal amount of inconsistency and achieved slightly lower CPU usage as a result. The consistency was kept within the default consistency bounds, so it is likely that if the bounds were even stricter, the Dyconit Consumer behave just as the Kafka Consumer.

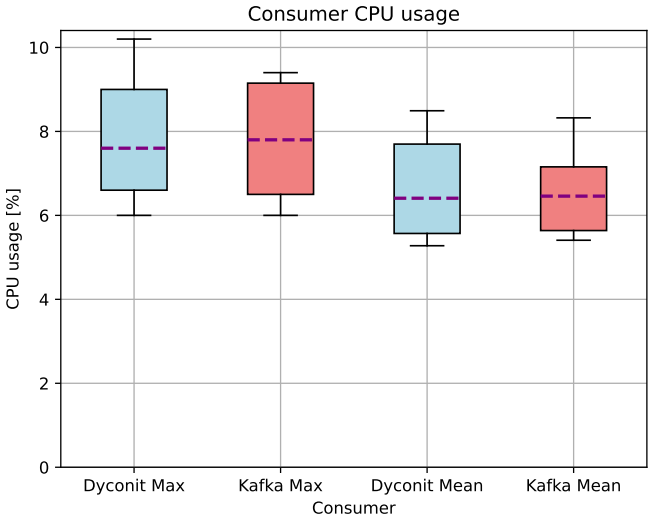

Figure 5.7: The effect of Dyconits in low workload scenario with topic priorities on CPU usage in E1.

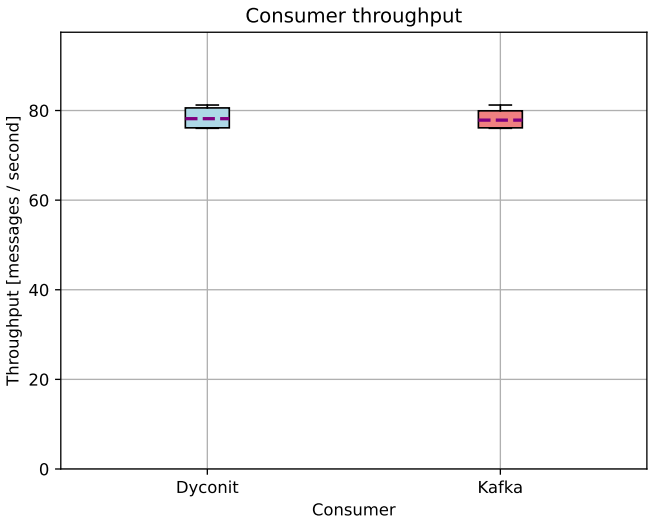

Figure 5.8: The effect of Dyconits in low workload scenario with topic priorities on message throughput in E1.

Figures 5.9 and 5.10 depict the performance of the system during **E2**. **E2** represents a high-workload scenario, and the data shows that Dyconits successfully reduced both the maximum and average CPU usage. The performance and consistency results together show that the Dyconit prototype allowed for some inconsistency which lowered the throughput and reduced the CPU usage.

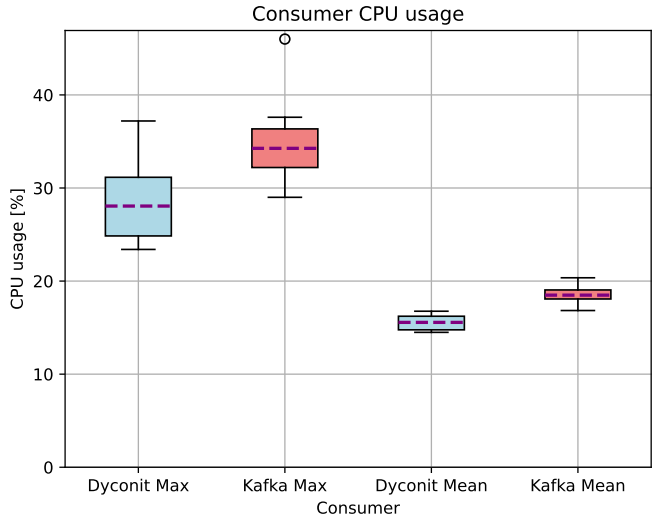

Figure 5.9: The effect of Dyconits in high workload scenario without topic priorities on CPU usage in E2.

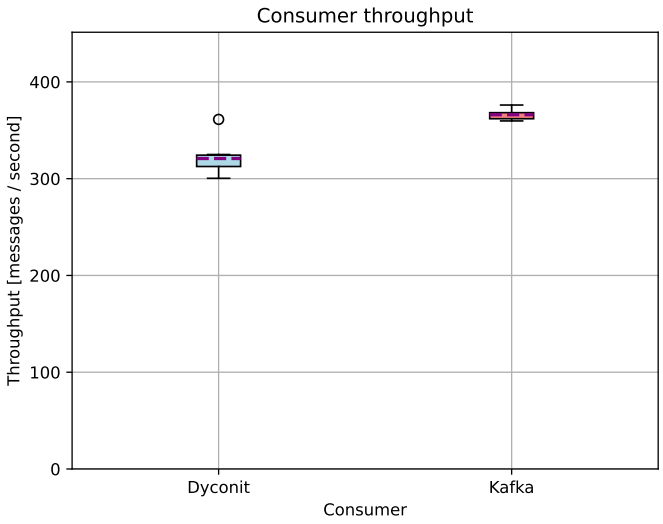

Figure 5.10: The effect of Dyconits in high workload scenario without topic priorities on message throughput in E2.

The average CPU usage is 18.1% lower, and the peak usage is 15.03% lower. Inspecting the data reveals that the decreased average CPU usage is primarily due to fewer and generally smaller usage peaks. The Dyconits allow for some inconsistency, so I expect the throughput to decrease. Figure 5.10 confirms the expectations and shows that the throughput decreased by 12.37%.

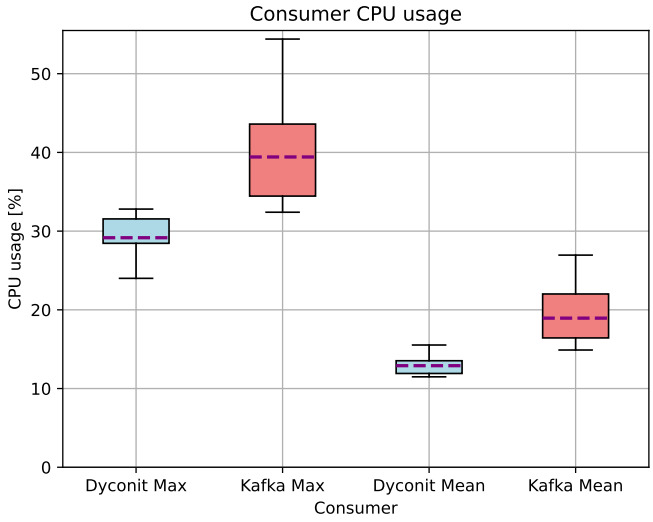

Figure 5.11: The effect of Dyconits in high workload scenario with topic priorities on CPU usage in E3.

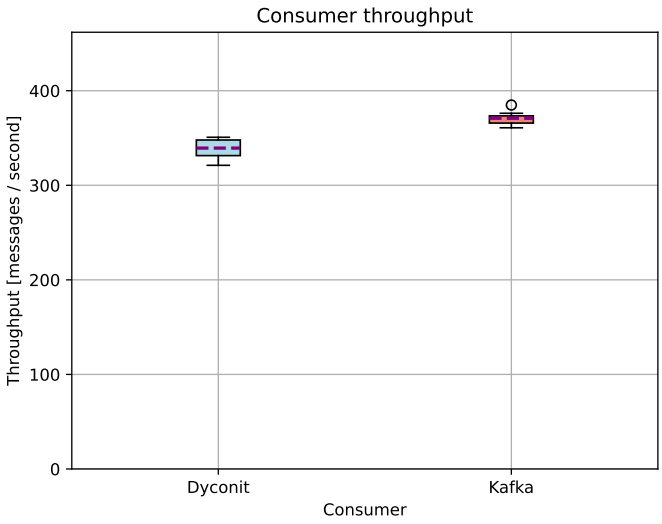

Figure 5.12: The effect of Dyconits in high workload scenario with topic priorities on message throughput in E3.

Figures 5.11 and 5.12 show the performance of the system during **E3**. Both the average CPU load and the maximum load were smaller in the Dyconit implementation. On the other hand, the throughput of the Dyconit version was slightly lower than that of the Kafka version. The addition of topic priorities allowed the Dyconit prototype in **E3** to focus on the important topic and considerably reduce CPU usage. However, the consistency decreased

for other topics, and the message throughput was slightly lower.

The average CPU usage in **E3** is 26% lower, and the peak CPU usage is 31.89% lower. Once again, the average usage increase seems to stem from a smaller number of peaks in CPU usage. In figure 5.12, the Dyconits decreased the throughput as well, in the case of E3 the throughput went down by 8.51%.

5.6.3 Throughput overhead

The Dyconits cause no significant overhead throughput. The number of messages exchanged for Dyconits, and thus the overhead throughput, depends only on the report rate and time. The manager sends updates after a significant enough change in production rate and after any consumer update. Thus, the following formula gives the upper bound on the throughput overhead.

$$\frac{number\_of\_producers \cdot producer\_report\_rate + consumer\_report\_rate}{combinded\_production\_rate} \tag{5.1}$$

However, the overhead should be closer to the lower bound in a system with a stable production rate, such as in my experiments.

$$\frac{consumer\_report\_rate}{combinded\_production\_rate} \tag{5.2}$$

Experiment	Meassured Overhead	Upper bound	Lower bound
1	0.94%	0.95%	0.22%
2 & 3	0.07%	0.16%	0.04%

Table 5.2: Message Overhead.

Interstingly, in table 5.2, in **E1**, in a low workload scenario, the overhead is much closer to the upper bound than to the lower bound. This indicates that the specified threshold in production rate change for sending updates was set too low, and in a low workload scenario, even a small change in production rate could trigger it. However, the upper bound of the overhead still comfortably satisfies the requirement for low Dyconit communication overhead **(NFR1)** as it is less than 1%. In **E2** and **E3**, the overhead was closer to the lower bound, as expected, as the experiments had a stable workload.

5.7 Discussion

Due to the challenges of setting up a realistic environment with multiple nodes and different workloads, I did not test our system on a real setup or a setup similar to the ones used in real-life use cases. Integrating and running my prototype in a real scenario would surely prove difficult and create many challenges, such as network overhead. However, I believe that the results still demonstrate potential benefits that could be achieved when running my implementation in a more appropriate environment and indicate that this is a promising direction for more research and testing.

6

Conclusion

This project aimed to extend the concept of Dyconits to an event-driven publish-subscribe system. I reviewed relevant background information and formulated three research questions to accomplish this. Furthermore, I have described my contribution to the research in terms of design and implementation. This chapter concludes the project by answering the research question based on the project's results and findings and suggesting areas for future work based on my project's limitations.

6.1 Answering Research Questions

(RQ1) Design of the system: How to design a Dyconit system for publish-subscribe systems?

Based on requirements relevant to publish-subscribe systems, I designed a Dyconit system. The design has three main components: the Dyconit Manager, the Dyconit Consumer, and the Dyconit Producer. Together, the components manage the system's consistency level. My design can apply dynamic consistency levels to different parts of the system based on their performance and importance, achieving optimistic application-oriented consistency.

(RQ2) Implementation of the system: How to integrate a Dyconit system for publish-subscribe systems?

Based on the design, I present a working prototype of Dyconits in a publish-subscribe system. I used the Confluent Kafka library in .NET in combination with custom features and functionalities to develop and configure the Dyconit Consumer, the Dyconit Producer, and the Dyconit Manager capable of upholding the design's requirements, successfully translating the concept of Dyconits onto a publish-subscribe environment.

(RQ3) Evaluation of the system: How to evaluate a Dyconit system for publish-subscribe systems?

I evaluated my prototype with synthetic workloads and different topic priorities. Based on the various configurations, Dyconits can sacrifice the consistency level of lower priority topics to increase the consistency of higher priority topics by 50% while reducing the average CPU usage by 30%, reducing the usage peak by 25% and slightly decreasing the throughput by 10 In scenarios with equal topic priorities, Dyconits achieve reduced CPU usage by introducing a configurable level of inconsistency and lowering the throughput. In conclusion, Dyconits successfully leverage the performance and consistency trade-off in high workload scenarios by allowing for a bound level of inconsistency while lowering the CPU usage at the cost of slightly lowering the throughput and a minor message throughput overhead. This has several real-life implications; firstly, Dyconits *reduce the risk of system overload*, and lower CPU usage spikes can prevent crashes and prevent degraded performance. Secondly, the Dyconits lowered the average CPU usage; this may indicate *better scalability* . However, the Dyconits also decreased the throughput, especially the throughput of low-priority topics. More research and testing are required to understand the scalability potential of Dyconits fully. Lastly, a Dyconit system can handle spikes in data more effectively, maintaining consistent performance under varying workloads and focusing on high-priority events, indicating *better load management*.

6.2 Limitations and Future Work

My work demonstrates that Dyconits have the potential to improve the performance and consistency attributes of publish-subscribe systems under certain circumstances. However, my project has certain limitations due to its scope and time. This section highlights the limitations and suggests areas for future work to address them and move the research forward.

6.2.1 Design limitations and future work

The first set of limitations and future work opportunities stem from the design of the Dyconit system. Firstly, *the Dyconit manager servers as a central management unit.* This provides benefits in monitoring the system and the enforcement of consistency. However, the manager is also a central point of failure and represents a potential scalability and performance bottleneck as it alone manages every node in the system. Moreover, the manager also introduces an overhead associated with a need for one more node.

To address this, future work could look at different *fault tolerance and scalability improvement methods*. One idea would be to introduce multiple managers or to manage separate parts of the system and present a way for them to coordinate on overall system consistency. A different approach could be merging the Dyconit manager with a preexisting Kafka node, for example, the broker, to reduce the overhead of a new node to remove the need for a new node and make the manager easier to replace.

An even more ambitious idea to improve the design would be to use the fast-progressing capabilities of *Machine learning and Artificial intelligence*. AI could improve the dynamic management of consistency levels by making strategic and complex decisions based on the requirements of each application in different environments without the need to develop new complex decision-making algorithms for each use case.

6.2.2 Implementation and evaluation limitations and future work

The prototype implementation and evaluation share some limitations. I proposed and implemented a prototype that showed promising results in terms of dynamic and optimistic consistency for a publish-subscribe system. However, it is *limited by the development and testing environment*. My experiments were conducted on a single machine, which presents several limitations. Specifically, limited scalability due to the machine's limited performance and a lack of real-world conditions such as network latency and bandwidth.

To overcome this limitation and further progress the development of consistency models, my Dyconit design would ideally be *implemented and evaluated in a large-scale distributed setting* that closely mimics real-life scenarios or represents one directly. Integrating my prototype into a large-scale distributed system would likely cause many unexpected issues. However, a comprehensive evaluation of a more complex and distributed infrastructure that closely corresponds to the use cases of publish-subscribe systems would offer invaluable insight into the upsides and downsides of the current design of Dyconits. A better understanding of the behaviour of Dyconits would help improve their performance under different circumstances and in varying environments and would inform areas for future research.

References

[1] **Apache Kafka**. `https://kafka.apache.org/`. Accessed: 2024-04-15. 1, 19

[2] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 1 2001. 1

[3] Sam Newman. *Monolith to microservices*. O'Reilly Media, 11 2019. 1

[4] Paolo Viotti and Marko Vukolić. **Consistency in Non-Transactional Distributed Storage Systems**, 2016. 1

[5] 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**. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21. ACM, October 2021. 1

[6] 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. 1, 2, 6, 10, 14

[7] Haifeng Yu and A. Vahdat. **Combining generality and practicality in a conit-based continuous consistency model for wide-area replication**. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 429–438, 2001. 1, 6

[8] 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. 5

[9] Yijie Liang. **Analysis of the Video Gaming Industry**. In *Proceedings of the 2022 2nd International Conference on Enterprise Management and Economic Development (ICEMED 2022)*, pages 1146–1150. Atlantis Press, 2022. 7

[10] 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. 7

[11] 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**. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 389–400, New York, NY, USA, 2008. Association for Computing Machinery. 7

[12] 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. 7

[13] Vlad Nae, Alexandru Iosup, and Radu Prodan. **Dynamic Resource Provisioning in Massively Multiplayer Online Games**. *IEEE Transactions on Parallel and Distributed Systems*, **22**(3):380–395, 2011. 7

[14] K.-T. Chen, C.-C. Tu, and W.-C. Xiao. **OneClick: A Framework for Measuring Network Quality of Experience**. In *IEEE INFOCOM 2009*, pages 702–710, 2009. 7

[15] Peng Chen and Magda El Zarki. **Perceptual view inconsistency: An objective evaluation framework for online game quality of experience (QoE)**. In *2011 10th Annual Workshop on Network and Systems Support for Games*, pages 1–6, 2011. 7

[16] Elvis S. Liu and Georgios K. Theodoropoulos. **Interest management for distributed virtual environments: A survey**. *ACM Comput. Surv.*, **46**(4), mar 2014. 7

[17] Nuno Santos, Luís Veiga, and Paulo Ferreira. **Vector-Field Consistency for Ad-Hoc Gaming**. In Renato Cerqueira and Roy H. Campbell, editors,

Middleware 2007, pages 80–100, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 7

[18] Sasu Tarkoma. *Publish / Subscribe Systems: Design and Principles*. Wiley, 6 2012. 8

[19] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. **The many faces of publish/subscribe**. *ACM Comput. Surv.*, **35**(2):114–131, June 2003. 8

[20] **Apache Kafka**. `https://kafka.apache.org/documentation/#design_pull`. Accessed: 2024-04-20. 8, 14

[21] Jurre Brandsen. **Design, Implementation, and Experimental Evaluation of Hestia**. *Vrije Universiteit Amsterdam*, 9 2023. 8

[22] 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. 17

[23] **Kafka Topics and Producers FAQs**. Accessed: 2024-05-09. 22

[24] **PayPal Community Blog | Scaling KaFKa to support PayPal's data growth**, 2023. Accessed: 2024-06-15. 34

Appendix A

Reproducibility

A.1 Abstract

My artifact provides the source code of my prototype and scripts used to evaluate the experiments. I also gave instructions on compiling and running my prototype and generating resulting graphs and plots. My artifact is publicly available on a GitHub repository, accessible by the following link: https://github.com/atlarge-research/Dyconit_Kafka

A.2 Artifact check-list (meta-information)

- **Algorithm:** Dyconits Consistency model
- **Program:** Dyconit consistency for publish-subscribe systems
- **Compilation:** net7.0
- **Run-time environment:** Windows 10
- **Hardware:** Processor AMD Ryzen 7 5800H, 32GB RAM
- **Execution:** dotnet run
- **Metrics:** CPU usage, Numerical error, Staleness error, Message throughput
- **Output:** logged data for each node
- **Experiments:** as described in section 5.5
- **How much disk space required (approximately)?:** 1 to 10 MB
- **How much time is needed to complete experiments (approximately)?:** 5 minutes
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT

A.3 Description

A.3.1 How to access

My work can be accessed on GitHub.

A.3.2 Software dependencies

Prototype related dependencies

- Confluent.Kafka, version 1.9.3
- Microsoft.Extensions.Configuration, version 8.0.0
- Microsoft.Extensions.Configuration.Binder, version 8.0.1
- Microsoft.Extensions.Configuration.Ini, version 6.0.0
- Microsoft.Extensions.Configuration.Json, version 8.0.0
- Serilog, version 3.0.1
- Serilog.Sinks.Console, version 4.1.0
- Serilog.Sinks.File, version 5.0.0

Evaluation scripts related dependencies

- numpy
- matplotlib.pyplot
- argparse

A.4 Installation

Clone the prototype and evaluation scripts from GitHub.

```
git clone https://github.com/atlarge-research/Dyconit_Kafka.git
```

A.5 Experiment workflow

After cloning the repository you can do the following steps to run experiments.

1. Run the docker-compose file `docker-compose up -build`.
2. If you want to run additional measurements, run `docker-compose down` after each measurement

A.6 Evaluation and expected results

1. Run `python3 .`
   `plotMeasurments.py [name][number of measurment]` after each measurement.
2. Run `python3 .`
   `plotExperiment.py [name]` after 1 to 10 different measurements.

`[name]` has to be the same for the measurements and experiment you want to plot together on the second step.

A.7 Experiment customization

- Configure the `*.csproj` files in `nodes/C*` folders to configure the consumers.
- Configure the `*.csproj` files in `nodes/P*` folders to configure the producers.
- Configure the `*.csproj` files in the `nodes/M` folder to configure the manager.