# Total Order Broadcast for Synchronous Distributed Systems: A Time-Sensitive Networking Approach

**Gabriel Behrendt**

A thesis presented for the degree of
Bachelor of Science

Fakultät IV
Technische Universität Berlin
December 21st 2021

# Declaration

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 21/12/2021

Gabriel Behrendt

.................................................................................................

Unterschrift

# Deutsche Zusammenfassung

Rechenzentren unterliegen einer hohen Verbindungsauslastung, die einer der Hauptfaktoren für erhöhte Latenzen ist. Eine Umstellung auf synchrone Kommunikation innerhalb des Rechenzentrums würde verschiedene Vorteile mit sich bringen, darunter vor allem die Tatsache, dass keine Warteschlangen für Pakete mehr erforderlich wären und somit Verbindungsüberfüllungen vermieden werden könnten. Eines der Probleme bei der synchronen Kommunikation ist jedoch, dass ihre Leistung davon abhängt, wie genau sich das System koordinieren kann. Time-Sensitive Networking (TSN) Technologien haben in letzter Zeit viel Aufmerksamkeit von Netzwerkkartenherstellern erhalten und rüsten Netzwerkgeräte mit der Möglichkeit auf, die Distanz zwischen dem Betriebssystem und der Netzwerkkarte zu überwinden.

In dieser Arbeit entwerfen wir einen Prototyp eines Total Order Broadcast-Dienstes, der die Koordination mehrerer Maschinen in einer synchronen Rechenzentrumsumgebung ermöglicht und einen TSN-Traffic-Shaper auf der Kommunikationsebene einsetzt. Das Ergebnis ist eine Anwendung, die in der Lage ist, in begrenzter Zeit einen Konsens über einen Nachrichten-Commit zu erzielen, und die in keiner Weise von Netzwerküberlastungen betroffen ist. Wir stellen jedoch fest, dass die vom Betriebssystem verursachten Scheduling-Verzögerungen die Grenzen des Ansatzes aufzeigen, da sie einen Engpass in Bezug auf geringere Latenzen und höheren Durchsatz darstellen.

# Abstract

Data centers experience a high link utilization, which is one of the main contributing factors to tail latency. A switch to synchronous communication for traffic within the data center would result in various benefits, above all due to the fact that packet queues would no longer be required, therefore eliminating congestion. One of the issues with synchronous communication, however, is that its performance depends on how precisely the system can coordinate itself. Time-Sensitive Networking (TSN) technologies have received a lot of attention by NIC vendors recently, and provide network devices with the means to overcome the latency caused by the distance between the operating system and the network device.

In this thesis, we implement a prototype of a Total Order Broadcast service, which would enable the coordination of multiple machines inside a synchronous data center setting, and makes use of a TSN traffic shaper at the communication level. The result is an application that is able to reach agreement on message commitment in bounded time and is not affected in any way by network congestion. However, we find that scheduling delays introduced by the operating system illustrate the limits of the approach, as they are a bottleneck in regard to lower latency and higher throughput.

# Contents

# Chapter 1

# Introduction

The end of Moore's law [1] implies that, if data centers wish to continue to scale at the current pace, they must do this more than ever horizontally. Horizontal scaling, i.e. adding more machines to distribute tasks, requires more intra-data center coordination, introducing further load on already highly utilized networks. New approaches are needed to maintain low latencies and high throughput. Here, viable solutions can be found in improving networking devices by augmenting them with features that make them more independent from the underlying operating system (OS), i.e. kernel-bypass techniques [2].

At the same time, modern network interface cards (NICs) can be equipped with hardware that allows the implementation of the *Time-Sensitive Networking* (TSN) standards, providing the means for reliable real-time Ethernet communication. The networks where this technology finds its use, e.g. industrial networks, however, do share some similarities with data center networks. They are isolated networks, the topology is mostly static and traffic is structured. TSN technologies may not only be applicable to data center networks, they may also be an additional factor in favor of a change towards a more synchronous system model in data centers. Such a transition would solve a plethora of difficulties today's data centers must face [3].

The question arises if TSN technology and its integration in popular commodity OSs (e.g. Linux) is ripe and fit to be applied to a different scenario than the one it was originally conceived for. The scenario of choice to start finding an answer to this question is an application aimed at allowing for intra-data center coordination, a *Total Order Broadcast* service. In this thesis, we implement a time-division multiple access (TDMA) based Total Order Broadcast algorithm into a synchronous service that, rather than relying on the best-effort TCP/IP stack, applies the TSN *IEEE 802.1Qbv* standard to achieve reliable, bounded communication between endpoints. In addition, the prototype produced for this thesis acts as a first attempt of implementing a small fraction of a potential synchronous data center. The result is an application that is able to reach agreement on message commitment in bounded time and is not affected in any way by network congestion. Moreover, we also point out several challenges with regard to the real-time nature of synchronous computing that bring about the limits of the implemented algorithm, our approach and the underlying OS. Such challenges must be overcome for the synchronous solution to be deemed practicable.

The rest of this thesis is structured as follows. Chapter 2 examines the problems related to a best-effort service model and the benefits a synchronous approach would bring to data centers. The Total Order Broadcast problem is discussed and the TSN standards we already verged on are introduced in more detail. In chapter 3, various work that serves as a basis to this thesis is presented. Chapter 4 serves to showcase the approach taken in order to answer the research question. The Total Order Broadcast algorithm this thesis concerns itself with is discussed in chapter 5 and we present the resulting prototype in chapter 6. Chapter 7 analyzes noteworthy properties of the prototype through a benchmark. In chapter 8, we discuss the results and give an outlook into possible future work in relation to the original question and the synchronous data center. In conclusion, we sum up the most relevant findings and contributions of this thesis' work in chapter 9.

# Chapter 2

# Motivation

## 2.1 The Issue with Best-Effort

When the Internet was originally designed, it was done so with the priority in mind of allowing for robust communication that can continue to function in spite of failures and packet losses [4]. The result was a best-effort service model, which provided the flexibility needed to integrate different kinds of already present networks (e.g. Ethernet networks, which on their part already rely on the best-effort approach) into one network of networks. Needless to say, the architecture has been tremendously successful, so much that its ubiquity led to a variety of applications being integrated. As a consequence, many of today's distributed systems inherit the principles the Internet was founded upon and follow an asynchronous model, making as few assumptions about the network's topology and its traffic as possible [3, 5, 6].

Nevertheless, the best-effort service model comes with its flaws, too. As endpoints are effectively competing among each other to get a message across communication links, a network can struggle to handle high loads. Packets may experience longer queueing delays or even get dropped [7]. Although there are mechanisms to mitigate the problem (e.g. congestion control and bandwidth reservations), some types of applications still suffer from the variance in latency that results from it [6,8].

Particularly data centers are systems that experience high loads, given the oversubscribed nature of their networks [9]. Such systems must deal with high amounts of requests, and possibly also coordinate internally to handle them. While most communication will still function within the expected delays, occasional higher latencies will occur. This *tail latency*, i.e. the large delays caused by the variance in latency partly resulting from the best-effort approach, can work as a bottleneck in data centers. When requests are handled in a joint effort by multiple nodes within the system, the overall response time is heavily affected by the slowest link [10, 11]. As low response times directly correlate to traffic (and therefore revenue) [12], finding ways to mitigate tail latencies in order to provide a more predictable service carries crucial significance.

## 2.2 A Synchronous Approach

While there are different factors contributing to tail latency [11], the issues involving the rather unique characteristics of data center networks (and rate of packet drops resulting from them) have caused a wide range of research concerning the control of traffic within the network (a survey thereof can be found in [13]). A suggested approach to avoid long queues and packets being dropped is the introduction of techniques to achieve deterministic communication [6, 9]. A proposition by Yang et al. pushes the notion even further. In their paper "The Synchronous Data Center" [3], the authors advocate to abandon the asynchronous model altogether in favor of a synchronous one[1], where all actions would be scheduled a priori. It may appear controversial, but the advantages of such a solution considerably go beyond the reduced and more predictable latencies.

For instance, a synchronous model would enable the deployment of synchronous algorithms, which in theory are inherently more efficient than their asynchronous counterpart [14]. This is due to the fact that in such an algorithm, time can carry a meaning and can therefore be used as a computational tool [15]. Furthermore, combined with the assumptions that a synchronous solution allows for, this often results in algorithms whose implementation is much simpler than the one of their asynchronous counterparts [16].

In their paper, Yang et al. also claim that, in order to achieve the objective of a fully synchronous data center, inspiration must be taken from the cyber-physical systems (CPS) domain. After all, safety-critical systems can cope with non-determinism even less than data centers, this being the reason why the field has been developing synchronous systems for decades. It is therefore reasonable to keep an interest in technologies and approaches that have their origins in the domains of CPS, embedded systems and real-time systems.

## 2.3 Total Order Broadcast

Migrating data centers from a synchronous to an asynchronous model would certainly require an enormous effort. One of the many hurdles would be redesigning approaches for tasks which have been tackled from an asynchronous perspective until now. Among these is Total Order Broadcast, a communication primitive that can provide the basis for the implementation of data center protocols [17, 18]. It provides certain guarantees allowing for assumptions that often yield simpler algorithms. This section will introduce the problem, discuss why it is relevant for the data center setting and, finally, present an approach to solve the problem in a synchronous fashion.

The Total Order Broadcast primitive defines two actions that a node taking part in the system can perform. One is the *broadcast* action, in which a node sends a message to all other nodes in the system[2]. The second is the *deliver* action, which can be understood as receiving and—perhaps after processing it—accepting a message that was previously broadcast.

The requirements an algorithm should fulfill to provide Total Order Broadcast in its most basic form are the following [20, 21]:

[1]The main difference between a synchronous and an asynchronous system lies in the assumptions made about processing and communication times. In a synchronous system, they must be bounded and the bound must be known.

[2]Or a group thereof. In that case, the communication is referred to as a *multicast* [19].

I *(Validity)* If a correct process broadcasts a message $m$, it eventually delivers $m$.

II *(Uniform Agreement)* If a process delivers a message $m$, then $m$ is eventually delivered by all correct processes.

III *(Uniform Integrity)* A message $m$ must be broadcast before it can be delivered by a process and it can be delivered at most once.

IV *(Uniform Total Order)* If two processes $p$ and $q$ both deliver messages $m$ and $m'$, then they do so in the same order.

Properties II-IV may be costly to provide under some circumstances, which is why they are sometimes relaxed to nonuniform properties, meaning only correct processes are assumed to comply to them [19].

Notice how the primitive allows for *reliable* communication within a group of nodes, with the added property that messages are delivered in total order. This allows for coordination in a distributed system involving multiple nodes and the possibility to take decisions based on the order of messages. More specifically, the primitive is powerful when trying to solve agreement problems.

In order to substantiate the utility of Total Order Broadcast in a data center environment, we will briefly introduce two example cases that are relevant for data centers where the primitive can come into aid.

### 2.3.1 Examples

**Replication.** Replication of data can aid a distributed system in improving its reliability. In case a node $n$ crashes, its failure can be masked by another node that holds the same pieces of information needed to handle the requests that $n$ is responsible for [22]. Further, replication can also help improve the performance of a system such as a data center, as it allows to distribute the work load among other machines by virtue of horizontal scaling [22]. One way to replicate write operations is the state machine approach (also known as active replication) [23], where the idea is that if updates are applied in the same order on all nodes, consistency will eventually be achieved. A Total Order Broadcast service significantly facilitates the implementation of such replication. Nodes wanting to perform a write operation merely have to broadcast the change to the nodes they wish to replicate to and the service will ensure the updates are delivered in the correct order.

**Distributed transactions.** Transactions are usually required to be atomic, meaning either the transaction is successful and fully applied to the database or the database remains unchanged. In a distributed system such as a data center, a transaction may involve more than one node. Therefore, to ensure atomicity for the whole transaction, the nodes must agree on whether to commit on the operation or, in case at least one involved node cannot apply the changes, abort [22]. This atomic commitment problem can be solved through the two-phase commit protocol [24]. The protocol makes use of a coordinator to prepare the involved nodes for the commit and, should all the nodes communicate that they have prepared successfully, the coordinator commands to commit the changes.

However, a problem emerges when the coordinator crashes before it can take a decision on whether to commit or not, since the nodes cannot make a final decision until the coordinator recovers. A fault-tolerant variant of two-phase commit exists to circumvent this issue [25]. In this variation, nodes broadcast whether they can go through with the operation to all other involved nodes rather than just messaging the coordinator, therefore allowing them to determine the commit decision independently. Furthermore, to tolerate the failure of other nodes, an abort decision in stead of other nodes can be broadcast if a crash is suspected. Should the node in question not have failed and broadcast a message itself, the first decision concerning the node must be considered. Total Order Broadcast ensures that, in case more messages were sent concerning one node, all the nodes receive the messages in the same order and can thereby come to an agreement on whether to commit or abort.

### 2.3.2 Synchronous Total Order Broadcast

The previous subsection gave an intuition on how valuable Total Order Broadcast can be in a data center setting. The implication is that, should a data center make use of Total Order Broadcast, a synchronous data center would on its part also require a synchronous implementation of the primitive. Although there are various ways to provide Total Order Broadcast in a synchronous fashion [19], for this thesis, we will focus on an implementation based on TDMA cycles [19,26]. Such an algorithm defines the actions a node must perform within predetermined periods of time, such that the approach fits perfectly to the notion of a synchronous data center, where recurring tasks can be scheduled according to their deadline.

The approach of the solution is based on the idea that access to the communication media is seen as a timed privilege that only one node at a time can hold. Within a cycle, messages are therefore broadcast in an order known a priori by all partaking nodes. Since the receiving nodes know exactly in which time frames to receive the messages from which nodes, the messages can be delivered in FIFO order. In absence of failures, this method alone already yields a Total Order Broadcast service. To make it fault-tolerant, an underlying membership service can keep track of the nodes functioning correctly, such that, should a message be correlated to a failure, this can be detected and the message is not delivered (an actual implementation of the approach is discussed in chapter 5).

The requirements for such an algorithm are similar to the ones found in every synchronous algorithm. The TDMA strategy requires that messages are received within a known time frame, as receiving nodes are programmed to expect a message neither earlier nor later than that interval. Should a node not be able to perform its communication within the predetermined time slot, it is to be deemed as faulty. Deterministic networking should provide this property by ensuring that if a message is sent out in a timely manner it also arrives at the destination before its deadline. Moreover, time synchronization is necessary, as the only basis the nodes cooperate upon is time. The synchronization should be as precise as possible, as additional precision allows nodes to perform tasks more efficiently and an imprecise synchronization may result in actions being performed uncoordinated. Finally, the algorithm can only rely on TDMA if there is a traffic control mechanism that ensures that faulty nodes cannot corrupt other correct nodes by sending messages that collide with other ones.

## 2.4 Time-Sensitive Networking

The TSN standards[3] aim at making communication through Ethernet deterministic so that systems hosting applications with real-time requirements can still benefit from the flexibility and ubiquity of traditional Ethernet. In part, these hardware standards define features that bridges and other networking devices should equip in order to provide precise time synchronization and traffic shaping, which leads to bounded latency, low jitter and no frame loss due to congestion. Of particular interest is the IEEE 802.1Qbv standard, which specifies the TDMA based Time-Aware Shaper. This standard ensures high levels of determinism by enforcing strict coordinated access to the communication media, even allowing for applications with hard real-time requirements [27].

The TSN standards are especially suited for networks where timeliness is crucial, such as audio/video processing, automotive and industrial networks [28]. However, the usefulness of TSN should not be constrained to its obvious use case. While a technology being specified by an IEEE standard is not necessarily always a recipe for success [29], TSN is currently one of the focus points of NIC vendors[4]. TSN technologies might soon find their place in all kinds of networking devices, making the standards attractive for other applications. Indeed, the deterministic networking, precise time synchronization and TDMA based channel access that can be provided through TSN perfectly fulfill the requirements needed to implement a TDMA based synchronous Total Order Broadcast protocol.

Consequently, as TSN is one of those real-time technologies that are of interest in the path towards a synchronous data center, TSN will play a key role at the networking layer of the implementation this thesis aims to discuss.

[3] `https://1.ieee802.org/tsn/`, accessed December 8th 2021

[4] IETF 105 Technology Deep Dive: How Network Interface Cards (NICs) Work Today `https://www.youtube.com/watch?v=wHM7RVk3-yk`, accessed July 23rd 2021

# Chapter 3

# Related Work

To our knowledge, so far there has been no research concerning the possibility of employing TSN to allow for deterministic networking in a data center setting. However, the solution presented in this thesis is only made possible by extensive research and progress across different domains. This section serves to give an overview about the main contributing literature published in what appear to be three entirely separate fields.

## 3.1 Data Center Networks

The complications relating to tail latency introduced in chapter 2 have prompted a wide range of research that aims at mitigating the problem in order to improve performance in data centers [6, 11–13, 30]. Additionally to the previously mentioned traffic control survey [13], a further introduction to the topic is provided by Althoubi et al. [11]. The authors name various sources of tail latency in a data center network and propose three separate solutions.

Perry et al. [6], however, propose an alternative solution to the problem with a system where all communication is scheduled by a single arbiter that distributes the routing plans and the permission to access the communication media to the nodes in the system. The approach shares some similarities with the one this thesis wishes to discuss. Their *Fastpass* system effectively achieves deterministic networking by reducing queue sizes by a factor of 240. The authors also state that a possible improvement to their system can be achieved through networking hardware that allows packets to be sent out at a specific point in time, therefore reducing delay and jitter caused by the operating system. The prototype implemented for this thesis uses such hardware for this exact purpose.

The results attained by the Fastpass system are one of the contributing factors for the aforementioned proposition by Yang et al. [3]. It is a recently (2019) published conference paper which discusses the benefits of migrating data centers from an asynchronous to a synchronous model. The importance of the paper in relation to our work has already been discussed extensively in section 2.2.

## 3.2 Total Order Broadcast

Total Order Broadcast is an older, very much established topic in the distributed communication research community. Plenty of literature has been published introducing algorithms [26,31], relating the problem to other known problems [21,32] and comparing different approaches to the problem [19]. Most notably, Defago et al. [19] have generated a survey of numerous Total Order Broadcast algorithms, classifying the properties and the methodologies.

Other research has shown how Total Order Broadcast can assist in solving prominent problems with regard to distributed systems. The fault-tolerant two-phase commit algorithm based on Total Order Broadcast [25] and Lamport's state machine approach for replication [23] have been discussed in chapter 2. Further Total Order Broadcast techniques for the distributed commit problem and replication are presented respectively in [33] and [34].

## 3.3 Time-Sensitive Systems

As already discussed in chapter 2, any research concerning the methodologies of building deterministic systems is of interest on the path towards a synchronous data center. In that regard, Hermann Kopetz' chair for Real-Time Systems[1] has done a considerable amount of work that is relevant for this thesis. The Total Order Broadcast service that is implemented in this thesis has its roots in the membership service [26] designed for the chair's Time-Triggered Protocol (TTP) [35]. More generally, TTP is of significant interest with regard to a synchronous data center. It builds the foundation for a system[2] where the processing of tasks is mostly dependent on the passage of time rather than events. Further, the chair has developed the Time-Triggered Ethernet extension [37], which can be regarded as a predecessor of the TSN standards, as they share similar goals and methods.

An introduction to the TSN standards that provide the deterministic networking needed to implement the synchronous Total Order Broadcast algorithm, is provided by Messenger [28] and Finn [38]. Additionally, the IEEE 802.1Qbv standard and the therein defined Time-Aware Shaper are discussed by Steiner et al. [39].

[1]Vienna University of Technology

[2]the Maintainable Real-Time System (MARS) [36]

# Chapter 4

# Approach

The goal of this thesis is mainly to explore a new application of TSN technologies to a data center environment. The notion is extended to the idea of a synchronous data center, a setting where we believe TSN's full potential can be unleashed. Through our results, we wish to contribute to the dialogue that Yang et al. [3] meant to initiate. The work of this thesis consists of both a theoretical and a practical part that are presented in the following sections.

## 4.1 Theory

Before a prototype can be built to explore a solution, an appropriate synchronous Total Order Broadcast protocol must be selected. Kopetz et al. [26] define a TDMA membership service that can be transformed into a Total Order Broadcast protocol with few adjustments. The resulting algorithm, however, has never been thoroughly discussed. We believe that an implementation of the prior can only be reasonable on the basis of a more detailed study. Therefore, in this thesis, we first describe the membership service and how to convert it to the Total Order Broadcast that is of interest. Following that, the algorithm is analyzed in terms of its correctness, the type of Total Order Broadcast it provides and its behavior in presence of different kinds of failures. Finally, we consider further properties such as latency, message overhead and bandwidth usage and discuss the suitability of the algorithms in terms of the purpose of this thesis.

## 4.2 Implementation and Evaluation

The practical part of our work consists in applying the algorithm to the end of creating a prototype that makes use of TSN technologies allowing deterministic communication. With consideration to the idea of a synchronous data center, we attempt to explore a prototype for a system running on top of a mainline Linux OS. We present the methods we follow to implement the TDMA protocol, while also presenting solutions to issues that are not addressed by the original algorithm itself. These include a concrete procedure to determine whether a node's failure has verified, providing a service interface for higher layers and determining methods to deal with the lacking real-time guarantees provided by the OS.

Following that, we present the results yielded by a static analysis and various benchmarks, where we compare the performance of the TSN solution and one based upon traditional best-effort Ethernet. The evaluation considers properties such as the mean time between failures (MTBF), latency in terms of message communication and delivery and throughput. Further, we discuss limits of the approach, its practicability and possible improvements, especially with regard to a (possibly synchronous) data center environment.

# Chapter 5

# Algorithm

## 5.1 Membership Services and the MARS Algorithm

Awareness about the state of nodes that are supposed to be cooperating plays a key part in the design of fault-tolerant systems. A node can be called part of the membership when it is active. Consequently, a membership service is tasked with reaching agreement on which nodes are still participating correctly. In a safety-critical real-time system it is critical that component failures are detected and communicated to other nodes within bounded time [5]. The MARS membership service [26] achieves this by relying on cycles during which each member has the opportunity to confirm it is still functioning correctly. Although originally meant for the domain of hard real-time applications, this synchronous algorithm follows the approach presented in section 2.3.2 and is able to provide the bounded failure detection needed to attain fault-tolerance within a TDMA Total Order Broadcast protocol. In the following, we will describe Kopetz et al.'s algorithm in further detail as it is closely related to the Total Order Broadcast implementation this thesis concerns itself with. A more formal description of the MARS membership service can be found in Kopetz et al. [26].

Kopetz et al.'s algorithm is based on the idea that each node $n$ has a *membership point* once per cycle, right before it is supposed to send a message. At $n$'s membership point, every node (including $n$) inspects the behavior of $n$ during the last cycle and also how other nodes have perceived it. This allows the correct nodes in the communicating group to reach agreement on whether $n$ had been a member at the moment it sent its last message (we refer to this moment as the *membership decision*).

In order to make the membership decision, the involved nodes maintain both a *view vector* and a *view matrix*. The view vector consists of one boolean entry per node and maintains information about the activity of other nodes within the last cycle. Since the order in which sending slots are assigned is known a priori, each node knows at all times from which other node to expect an incoming message. The view vector keeps track of whether the last expected message from a node was received or not. In its sending slot, $n$ attaches its view vector to the message. Upon receiving, the other nodes update their own view vector and attach $n$'s view vector to their own view matrix. At the end of the cycle, each node has a view matrix consisting of the most recent view vectors of the group.

The algorithm makes strong assumptions about the behavior of nodes. The so-called *selfchecking node* the algorithm relies on either operates correctly, producing valid results, or does not operate at all. Nodes should therefore themselves be able to detect their own failures at the membership points and, should that be the case, shut down independently. Furthermore, sending nodes broadcast their message twice, which reduces the risk of messages not being sent out. This and further experiments prompt the authors to stipulate the fault hypothesis, i.e. an assumption about the types of possible failures, that only one failure may occur within two TDMA cycles. These assumptions ease the detection of failures significantly and allow for a simpler, faster membership service.

The assumptions imply that, in the case of a missing message, a look into the view matrix suffices to determine who is responsible: should none of the nodes have received the last message from the sender, the sender is at fault. Should, on the other hand, another node have received the message from the sender, the receiver must be at fault. In both cases, should it still be running, the faulty node is able to recognize the failure independently and switch off immediately (the handling of such failures in discussed in further detail in section 5.3.2).

Being a TDMA cycles based algorithm, the requirements for the implementation of the protocol are the same as mentioned in section 2.3.2.

## 5.2 From Membership to Total Order Broadcast

The MARS membership service can be adapted to allow the set of cooperating nodes to perform cooperated actions as well with a few alterations, effectively resulting in a Total Order Broadcast service. Kopetz et al. [26] state that, having established the membership at a given point, all nodes are aware of which messages have been received by which nodes within the membership set. In case a message was received by all members of the group, agreement can be found on performing the action, as every correct node will deliver the message. In the other case, i.e. if some message was not received by a node in the set, the message must be eliminated by every node. In other words, the membership point of node $n$ is also the moment in time on which the cooperating nodes decide whether to deliver or discard $n$'s last message.

## 5.3 Properties

While the resulting Total Order Broadcast algorithm is to be accredited to Kopetz et al. [26], the paper only briefly mentions the algorithm (which we will refer to as *MARS Total Order Broadcast*, or just the MARS protocol) in relation to possible applications of the MARS membership service. The algorithm is included in Defago et al.'s survey [19], where it is classified among the privilege-based algorithms (the right to send a message is seen as a privilege that is transferred from node to node through the system dependent on the time), but not analyzed thoroughly. Therefore, in the following, we will discuss properties that the MARS Total Order Broadcast protocol presents.

### 5.3.1 Type of Total Order Broadcast

In order to confirm the algorithm's correctness and understand the type of Total Order Broadcast it provides, it is sensible to review the properties listed in section 2.3 and see if, and how, the MARS protocol fulfills them.

The first property, i.e. validity, is respected. A correct node $n$ sends its message during its sending slot. From the correctness of $n$ it also follows that the node would continue running until its upcoming membership point. There it would conclude that, since the message it sent cannot be associated to any failure, the message is to be delivered.

Also the property of uniform agreement is respected. The assumption concerning selfchecking nodes ensures that even a faulty node only delivers valid results. As a consequence, if any process delivers a message, it is the correct decision and therefore all the other correct nodes would also conclude that they must deliver the message in question.

The design of the protocol ensures that the only messages that are taken into consideration for delivery are messages that were broadcast by some node in the previous cycle. Furthermore, a message is only taken into consideration for delivery once, at the subsequent membership point of the sender node, meaning it can only be delivered once. The MARS protocol therefore achieves uniform integrity.

As pointed out in section 2.3.2, total order is given by the nature of TDMA communication, i.e. the globally a priori known sequence of senders and understanding of time. Membership points happen in a strict order that is maintained by every node; therefore message delivery is also done in that same order. In fact, the algorithm not only provides uniform total order, it also provides FIFO ordering, as the order in which the messages are sent out corresponds to the order in which they may be delivered.

Since all the properties for traditional Total Order Broadcast and the additional one of FIFO order are applicable to the protocol, it can be concluded that the MARS Total Order Broadcast service allows for uniform FIFO totally ordered broadcasting.

### 5.3.2 Fault-Tolerance

The previous subsection shows how the MARS protocol does indeed provide Total Order Broadcast. However, implementing Total Order Broadcast that only works as long as every node does exactly what it is supposed to is not considered hard [19] and yields a rather unfunctional algorithm. Possible failures should always be held in consideration, especially in the domain of data centers, where the presence of more machines directly translates into a higher probability to encounter failures. We shall therefore introduce the most common types of failures as described by Van Steen and Tanenbaum [22][1] and analyze the behavior of the MARS Total Order Broadcast service in presence of the prior.

**Crash Failures.** In the context of the algorithm, a crash failure implies that a crashed node $n$ is unable to deliver the messages that it had received within the cycle prior to the failure. As $n$ was still an active member on its most recent sending

[1]with the exception of response failures, which are not directly applicable to the protocol.

slot, $n$'s last message would still be delivered by the surviving nodes. In its following sending slot, however, $n$ would remain silent, prompting the other nodes to exclude $n$ from the membership set. It is important that after crashing $n$ remains silent for at least the duration of two TDMA cycles, such that the crash is definitively recognized by the other nodes. After that, $n$ can rejoin.

Notice how, although $n$ is not able to deliver its own message, none of the Total Order Broadcast properties are violated. The validity property requires that *correct* nodes deliver their own messages. Due to its crash, $n$ is not a correct node, hence the property is still respected. And, since $n$ worked correctly up until it crashed, none of the other properties can be violated either. The conclusion is that the MARS protocol can tolerate this type of failure.

**Omission Failures.** With respect to the MARS protocol, an omission failure can be traced back to a problem with the communication links. Either the message was not sent out correctly or it was not received correctly. Kopetz et al. [26] distinguish between *outgoing link failures* and *incoming link failures*.

Regarding the former, the fact that messages are sent out twice per slot significantly reduces the probability of such an omission failure affecting the system at all. Should a node $n$ still be unable to reach other nodes with its messages, the other nodes will notice this and set their view vector accordingly. During $n$'s next membership point, $n$ will be able to figure out the problem by taking a look at its own view matrix and shut down accordingly without delivering its own message.

Incoming link failures are, for the most part, similarly easy to spot. Should $n$ fail to receive more than one message within the duration of a cycle, the fault hypothesis suggests that such a failure must have occurred for $n$. If $n$'s single incoming link failure occurred before the preceding node $k$'s sending slot, $n$ will be able to deduce its failure during its own membership point. This is due to $k$'s view vector showing that the message that $n$ failed to receive was in fact sent out and received by other nodes. In both these cases $n$ would recognize the fault within the end of the cycle and shut down. The case where it is $k$'s message that gets lost is, on the other hand, slightly more complex. During its own membership point, $n$ has no base on which to decide whether the fault is to be accredited to $n$ or $k$, meaning $n$ must assume it is still part of the membership set and send out its message. However, in the following cycle, it will become evident that an incoming link failure occurred at node $n$. Thus, $n$'s last message will not be delivered by any node and $n$ turns itself off.

Undoubtedly, the handling of omission failures in the protocol heavily relies on the fault hypothesis. However, as long as the fault hypothesis holds, MARS Total Order Broadcast can tolerate this kind of failures.

**Timing Failures.** Two types of timing failures can occur. A scheduling delay would result in computations being performed in an untimely manner, therefore potentially desynchronizing a node from the rest. It is up to the implementation to ensure that such a failure is eventually detected by the node, prompting it to shut down.

Otherwise, a timing failure occurs if a message arrives too early or too late. In a TDMA based algorithm, a message can only be received if it is sent during the appropriate time slot. Should that not be the case, the message is discarded.

Thus, the MARS protocol can treat timing failures identically to crash or omission failures, as long as the implementation provides the means for it.

**Byzantine Failures.** Ensuring that no property is violated in the presence of an arbitrary failure is not a trivial task. The assumption the authors of the MARS protocol make regarding selfchecking nodes is incompatible with byzantine behavior. The algorithm therefore works under the assumption that there is another present mechanism in the system such that such a failure cannot occur. For simplicity's sake, we are going to ignore this type of failure. It should be noted, however, that in a synchronous system byzantine fault tolerance can already be achieved if more than half of the nodes function correctly. Asynchronous byzantine fault tolerance, on the other hand, requires two thirds of the nodes to not fail [40]. Also here a switch to synchrony can be regarded for the better.

### 5.3.3 Performance and Suitability of the Algorithm

Now, after having established that the MARS protocol is correct and provides a certain degree of fault-tolerance, we will discuss several aspects that determine the quality of the algorithm and that determine whether the algorithm is an adequate option for the implementation pursued by this thesis.

One of the most favorable properties of the algorithm is the overhead it produces in communication. The view vectors that sending nodes append to their messages consist of a boolean value for every node in the system, meaning one byte can carry information about the perceived state of eight nodes. Kopetz et al. [26] set the message length to a fixed 1500 bytes (the maximum length of the payload of an IEEE 802.3 Ethernet packet). As a consequence, up to 120 nodes would have to be partaking in the protocol for the view vector to occupy just 1 % of the message length. Even if, depending on the implementation, it may be required to occupy some additional space for possible flags or alike, the overhead produced by the algorithm is minimal.

A further benefit provided by MARS Total Order Broadcast is the deterministic latency achieved both in message delivery and failure detection. Consider the case where a failure in node $n$ occurs right after its sending slot. While other nodes would not notice the failure during $n$'s upcoming membership point, $n$ would fail to send a message in its next sending slot. Even in this scenario constituting the worst case, $n$'s failure would be detected by all correct nodes within the duration of two TDMA cycles. A bounded error detection may lead to faster repair times which, in turn, can lead to higher availability of a system [5]. Regarding message delivery, both the latency and the variance thereof are lower. If a message is delivered, it is delivered in the upcoming membership point of the sender. As a consequence, a message is always delivered within the duration of a cycle, more precisely within the bounds of the duration of a node's membership point. This property is especially welcome in regard to the problem concerning tail latency discussed in chapter 2. Consider a situation where a client makes a request that reaches node $n$ within the system. Node $n$ could use its upcoming sending slot to communicate the request to the rest of the system and would know within a set duration of time whether the message was delivered or not. This determinism is already a big step in the direction towards the goal of reducing tail latencies.

On the other hand, the protocol exhibits grave shortcomings in terms of bandwidth usage that need to be addressed. By nature, TDMA—as any other channel division mechanism involving static allocation—will always be less efficient than a best-effort communication where all nodes compete for access to the same channel [29]. This issue is elevated in the presence of a failure, where a node is unable to use its sending slot and therefore the full allocated bandwidth is wasted. A service relying on traditional best-effort Ethernet would not encounter such problems. As the algorithm is originally designed for synchronous safety-critical systems, the authors must have deemed the tradeoff to be worth it. In applications like a data center, however, it may be more difficult to justify the obvious waste of bandwidth.

Furthermore, to function correctly, the algorithm makes plenty of assumptions. The clock synchronization and deterministic communication can be provided, and the fault hypothesis, while it may appear optimistic, is supported by lengthy experiments described by Kopetz et al. [26]. However, ensuring that the assumption concerning selfchecking nodes is never violated is problematic, especially with regard to byzantine failures.

Nevertheless, the TDMA approach the algorithm follows fits perfectly to the idea of a synchronous data center where everything is scheduled. It is another example demonstrating how ideas belonging to the CPS domain can be repurposed for a different use and, being featured by Defago et al. [19], one might argue that it is the TDMA based Total Order Broadcast service with the most visibility. Certainly, if put up to comparison, MARS Total Order Broadcast would struggle to keep up with an established implementation of Total Order Broadcast. Regardless, more importantly, it comes with an advantage no asynchronous Total Order Broadcast protocol can provide: a simple implementation. As the goal of this thesis is to explore a certain solution and its viability rather than provide an implementation ready to be deployed, MARS Total Order Broadcast is a suitable protocol.

# Chapter 6

# Implementation

Now that we have defined and analyzed the algorithm our work is centered on, we can introduce the resulting prototype. The C programming language is employed for the prototype due to its performance, certainly a factor that cannot be disregarded when implementing real-time software, and ability to interact with the OS to perform actions such as altering a network socket's options with relative ease.

In this chapter, we introduce the approach we adopted during the implementation of the protocol and discuss several design choices concerning different aspects of the prototype.

## 6.1 Data Structures

The implementation relies on a number of data structures that aid in keeping track of the current state of a node within the protocol. This section lists the three most relevant ones. The *service time* data structure (table 6.1) aids in keeping track of a consistent view of the current cycle, turn and slice. Furthermore, each node stores an array of *node state* data structures (table 6.2) with an entry for each participating node. As this array holds the view vectors of all nodes, it also functions as the node's view matrix. Finally, the *service message* data structure (table 6.3) serves to communicate between the nodes.

The following sections refer to these data structures. The reasons why the implementation requires keeping track of certain information will be discussed therein.

## 6.2 The Protocol Cycle

The algorithm described in chapter 5 defines the following actions that a correct node must perform throughout the duration of a protocol TDMA cycle:

For each partaking node $n$, there is a membership point at which all nodes, including $n$, must decide whether $n$ is a member and their last message needs to be delivered. Subsequently, assuming no failures occurred, $n$ must broadcast its next message and the other nodes must receive it. A cycle is completed when this procedure was applied to all partaking nodes.

For the implementation, we opted to receive the cycle duration and the number of nodes as program parameters, to then split a cycle evenly among the nodes and

| | |
|---|---|
| **cycle** | count of completed cycles since the node started running |
| **cycle start time** | first timestamp taken within the current cycle |
| **current time** | latest timestamp taken by node |
| **current sender ID** | corresponds to the current turn within the cycle |
| **slice** | the current slice within the turn, either *MEMBERSHIP*, *SEND* or *PAUSE* |

Table 6.1: Service time data structure

| | |
|---|---|
| **node stage** | either *MEMBER*, *ACTIVE* or *INACTIVE* |
| **last msg** | last message that was received from this node |
| **node view vector** | if available, latest view vector of this node |
| **suspicion info** | both a flag to signal if a possible receiver omission failure is suspected and the ID of the node that triggered it |

Table 6.2: Node state data structure

| | |
|---|---|
| **node ID** | sender's identifier |
| **node view vector** | the sender's view vector (a single byte for eight nodes) |
| **stage** | current stage the sender is in |
| **suspicion info** | signals to other nodes that the sender suspects a failure, equivalent to the field in the node state data structure |
| **data** | actual service message the sender desires to find agreement on |

Table 6.3: Service message data structure

the tasks that need to be performed. We introduce the concept of a *turn*, a fraction of a cycle dedicated to one node's membership decision and its sending slot (meaning a node with the identifier $i$ is allowed to broadcast in turn $i$), in addition to a buffer period to allow all nodes to receive the broadcast message. In order to complete each of these actions within a turn, we split the duration of a turn again into three of the smallest protocol computational intervals, i.e. *slices*. A single slice's duration is therefore yielded directly by the formula

$$\frac{cycle\ duration}{3 \cdot number\ of\ nodes}.$$

With these constants at hand, a single running node can determine which actions to perform upon wakeup. Assuming all nodes are appropriately synchronized, each computation occurs in a coordinated manner.

The procedure repeatedly performed by the nodes is simple. Upon wake up, a node fetches the current system time, thereby calculates the current turn and slice, and saves the information into its service time data structure. Depending on these values, the node then performs the required computations and goes to sleep for the remaining duration of the slice. In the case of this prototype, the cooperating nodes are run on the same device, therefore receiving the current time from the very same clock[1]. As a result, the nodes stay synchronized and perform their computations at the appropriate time.

The following subsections go into detail about the computations being performed in the separate slices.

### 6.2.1 Node Entrance

To be able to deliver messages in a fault-tolerant manner, a node must join the membership. As noted by Kopetz et al. [26], a node trying to enter the protocol must first wait one cycle to collect the view vectors of all members and then broadcast its own view vector. At its upcoming membership point, the node is then able to verify if all members have received the message and can therefore join the membership.

The cycle and cycle time fields of the service time data structure (table 6.1) are used by the implementation to make sure that any protocol action except receiving is omitted in cycle 0. In cycle 1, a node $n$ can use its sending slice to communicate to the other nodes it wishes to join, changing its node stage in the node state data structure from *INACTIVE* to *ACTIVE*. In $n$'s upcoming membership slice, $n$ and all members that received $n$'s last message will verify if $n$ can join the membership. Should the view vectors of all members imply that $n$'s message was received, $n$ can join the membership, i.e. its stage is changed to *MEMBER*. Otherwise, $n$ shuts down.

There are, however, possible cases regarding a nodes' entrance that Kopetz et al. [26] do not cover. In our implementation, nodes communicate their node stage when broadcasting a message. This allows joining nodes to individuate members and, more so, recognize if a group of members has even been established yet. If, however, there is no established membership set, a node has no nodes to refer to when trying to enter the membership. This problem occurs whenever the protocol is started, since all nodes are at the *INACTIVE* stage. To allow for a simple creation of the first membership, the first node $n$ to be active may create the membership at its upcoming membership point. The only requirement is that there is at least another active node that received $n$'s message and confirmed this event through its view vector.

While this solution may potentially allow for faulty nodes to enter the first membership, this is an acceptable outcome. Unlike in Kopetz et al.'s membership service [26], the objective of this protocol is the ordered delivery of broadcast messages, and not necessarily keeping a consistent view of the correctly functioning nodes. Actual service messages are only broadcast and delivered when the membership count reaches three, as two nodes alone would not be able to reach an agreement in case of a failure. By the time $n$ would be able to deliver a message, there will be enough members to detect a possible failure by $n$, in which case $n$ will be removed from the membership set.

Furthermore, a new node's entrance may be delayed when the membership set consists of only two nodes. As no membership decisions are being performed[2], a

[1] A more realistic, distributed implementation would make use of IEEE 802.1AS, the TSN profile of the *Precise Time Protocol*, to synchronize the system clocks within nanosecond range.

[2] We do not define the membership entrance as a membership decision, as this would of course prevent a third member from entering at all. Two members may still agree on letting a third node join.

faulty node may remain a member and hinder the new node's entrance. To cope with such an occurrence, a node will shut down after a fixed number of cycles spent in a two node membership.

The next subsection covers how the membership decisions are implemented in the prototype.

### 6.2.2 Membership Decisions

For the purpose of this section we define the function $pred(n)$ to refer to the predecessor of a member $n$, i.e. the node that has its membership decision in the turn prior to $n$'s. The membership decision is taken at the beginning of a member's turn by each member. As already addressed in the previous subsection, this decision can only be taken if there are more than two members. In the case of a missing message, the sender and the receiver must be able to refer to a third view vector to be able to tell which of the two is at fault. To this end, our implementation ensures that the members keep track of the size of the membership set by adjusting the member count whenever a node gains or is revoked membership.

Before verifying the view matrix to detect a possible failure within the cycle leading up to the membership point, there is actually an additional step to be taken, i.e. resolving a possible receiver omission failure suspicion. As the concept will become much clearer when knowing what occurs during the membership decision, the description of the procedure is postponed to a later point of this section.

During a node $n$'s membership decision all members verify if a failure by $n$ transpired during the most recent cycle. The implementation checks for two possible kinds of failures: a sender omission and a receiver omission. For the sender omission, a receiver $m$ must verify through its own view vector if it received a message from $n$ during $n$'s last sending slice. In case a message was received, $m$ rules out a sender omission. Otherwise, it performs the same check as $n$. It iterates over all receivers view vectors and counts how many nodes did not receive $n$'s last message. If more than one node did not receive the message, both $n$ and $m$ come to the conclusion that $n$ had a sender omission.

The receiver omission check is only relevant for $n$. The node must verify whether, during the last cycle, some node $m$ sent a message that $n$ did not receive. If $n$ did not receive two or more messages from other active nodes, $n$ concludes its own failure per the fault hypothesis. Should only one message not have been received, $n$ checks which member was the sender. For any member $m$ other than $n$ and $pred(n)$, $n$ can verify if $pred(n)$ did not receive the message either and decide if the failure was due to $n$ or $m$.

As described in section 5.3.2, the critical case occurs when the missing message belongs to $pred(n)$. Here, $n$ goes into *suspicious mode*. It sets its own suspicion flag and remembers $pred(n)$, the node that caused it. If no other failure occurred and $n$ acts as a member for the following cycle, the subsequent *MEMBERSHIP* slice requires the suspicion check we previously touched upon. To this end, a member merely checks if $pred(n)$ is still considered a member. If the message that was not received by $n$ was caused by a failure by $pred(n)$, this sender omission will have been detected during $pred(n)$'s membership slice. Consequently, should $pred(n)$ still be a member, the failure detected in $n$'s previous membership slice is accredited to $n$.

There are two possible outcomes to the membership decision. Should a failure be accredited to $n$, be it through suspicion check, sender omission or receiver omission, $n$'s state is updated accordingly to the *INACTIVE* stage. The node $n$ itself shuts down, therefore not making use of its next sending slot. Thereby, other potential joining nodes deduce that $n$ is no longer a member and can also gain a consistent view of the membership set. The node $n$'s last message is discarded without being delivered. Should however no failure be accredited to $n$, the data of $n$'s last message is delivered by all members. This is also the case if a failure is suspected but not confirmed.

### 6.2.3 Service Interface

This section will cover how the implementation "delivers" a message and where these messages are originally taken from.

Our work focuses on the implementation of the MARS protocol with deterministic computing and communication. However, in order to act as a Total Order Broadcast service, an interface must be provided through which a user can feed some input messages to a node and verify if they are delivered. We opted for a rather simplified solution which involves one input and one output file for each node. It does not allow a user application to easily access the service's functionalities, but it does at the very least provide an abstraction of a higher layer, communicating to the prototype through an API.

While composing its next broadcast message and assuming there are at least three members, a member $n$ fetches a new line from its input file. The content of this line is interpreted as the message the node wishes to deliver. If the line is empty, this message only consists of the null byte. The message is stored by all members in $n$'s node state data structure. Should the members later, i.e. at $n$'s next membership slice, decide to deliver the message, it is appended to each member's output file.

### 6.2.4 Communication

We implemented multiple communication modes for the prototype. Besides communication based on TSN, there is the possibility to run a node through best-effort communication. This additional mode will be of special interest when evaluating the prototype (see chapter 7), but we will only focus on the details concerning the implementation of the TSN mode in this chapter.

In Linux, TSN traffic shapers are implemented through *qdiscs*[3]. A qdisc is an additional layer between the kernel and the network interface where packets are scheduled following some predefined rules. The Time-Aware Shaper, or IEEE 802.1Qbv, can be implemented by coupling the *taprio* and *etf* (Earliest TxTime First) qdiscs[4]. Listing 6.1 shows how to set the schedule for a single node in a system of three nodes. The first command sets the *taprio* qdisc defining two traffic classes. All packet priorities are mapped to traffic class 0, except priority 5 (chosen arbitrarily), which we are going to use exclusively for our ordered broadcast traffic. Priority 5 is therefore mapped to traffic class 1. We then set one entry for each of

[3]`https://linux.die.net/man/8/tc-prio`, accessed December 10th 2021

[4]`https://tsn.readthedocs.io/qdiscs.html`, accessed December 10th 2021

```
tc qdisc replace dev <interface> \
        parent root handle 100 taprio \
        num_tc 2 \
        map 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 \
        queues 1@0 1@0 \
        base-time <start> \
        sched-entry S 01 <slice_duration> \
        sched-entry S 02 <slice_duration> \
        sched-entry S 01 <slice_duration> \
        sched-entry S 01 <slice_duration> \
        sched-entry S 02 <slice_duration> \
        sched-entry S 01 <slice_duration> \
        sched-entry S 01 <slice_duration> \
        sched-entry S 02 <slice_duration> \
        sched-entry S 01 <slice_duration> \
        flags 0x1 \
        txtime-delay 50000 \
        clockid CLOCK_REALTIME

tc qdisc replace dev <interface> \
        parent 100:1 etf \
        clockid CLOCK_TAI \
        delta 50000 \
        offload \
        skip_sock_check
```

Listing 6.1: Setting the schedule for three nodes

the nine slices needed for a cycle in a system of three nodes, allowing only traffic of class 1 in the designated sending slices. While the qdisc will now make sure only packets with priority 5 are passed to the interface during sending slices, this alone does not utilize the i210's TSN capabilities fully. To ensure every frame is sent out in time, the *etf* qdisc is set with the offload parameter, such that packets leaving the *taprio* qdisc go through it. Finally, the `phc2sys` command is run in the background in order to keep the i210's hardware clock accurately synchronized with the system clock.

While this allows our packets to be transmitted in a timely manner, a distributed system making use of TSN needs to make sure that any device these packets encounter on their way to the destination does not delay or drop the packet. Ideally, this is achieved through TSN switches that are also aware of the schedule.

Upon startup, a node initialises a datagram socket and sets the protocol identifier to `0x88b5` (*Local Experimental Ethertype*) in order to send bare Ethernet packets. We set the socket priority to 5 and bind the socket to a VLAN interface, which will handle tagging the frame with the correct PCP field. The broadcast destination MAC address is also set; a sequence of six `0xff` bytes.

Once a cycle, a node's designated sending slice occurs. Here, the node must take the decision of what kind of message to broadcast or whether to broadcast at all. If the node still finds itself within cycle 0, it must wait the slice out. Otherwise, it must first create a buffer for the message and fill it with the necessary information (table 6.3). The data field is only filled if the node is a member and there are more than two nodes in the membership set. The complete service message can then be sent by passing it to the `sendto()` function, through which the socket, destination address and VLAN interface ensure a correct Ethernet frame is created. According to the algorithm, the `sendto()` operation is repeated twice within the sending slot.

Concurrently to one node's sending slots, the other nodes must ensure they receive the message that is potentially broadcast. To this end, a receiver polls the socket for a fixed duration which depends on the latency between the sender and the receiver. In case of a small slice duration, it is possible that a node broadcast happens close to the end of the schedule entry. In this case, a correctly sent out message may only arrive after the current slice has ended. The final slice of the turn, the *pause slice*, is provided to allow a node to complete the receiving operation without the risk of omitting the upcoming membership slice. Upon receiving, the sender's entry in the receivers' node state array is then updated with the values contained in the received message, or, should no message be received, the view vector is set to unknown and the node's stage is updated.

### 6.2.5 Real-time Behavior

As the node's base for cooperation is the current system time, the prototype is technically a real-time application. A number of design decisions have been made to deal with the fact that mainline Linux does not provide real-time guarantees.

To allow for timely wakeups, the protocol must be run with real-time priority. While in mainline Linux this does not ensure that every deadline is maintained, by doing this no non-real-time process will be preferred to our application by the process scheduler. Additionally, the final slice of a turn, i.e. the pause slice, aids in dealing with failures caused by a sender's possible delayed wakeup. Finally, in order to avoid disk operations to become a possible bottleneck regarding the slice duration, the two files that function as a service interface are kept in memory throughout a node's execution. Hence, as it stands, the input file cannot be modified at run-time by a user.

# Chapter 7

# Evaluation

In the following, the implementation of the prototype is first analyzed statically in regard to overhead, latency and link utilization. We then introduce the target platform and present the results that were yielded by various benchmarks in order to observe whether the practice confirms the theory.

## 7.1 Static Analysis

### 7.1.1 Protocol Overhead

Depending on the amount of nodes involved in the protocol, the total size of the headers of a service message (table 6.3) may differ. Since our target platform does not allow for more than four participating nodes, five bytes suffice. The node identifier, its view vector and its current stage all take up a single byte each and the remaining two are required for the eventuality of a suspected receiver omission failure. Such a header would occupy .3% of the total 1500 bytes that can form an Ethernet frame's payload, where the rest of the frame can be fit with messages that require delivering. While it may be desirable to utilize some of the 1500 bytes of payload for other layers of the network stack, and even with a larger amount of nodes involved, the protocol would still produce minimal overhead, which in turn contributes to better throughput.

### 7.1.2 Latencies

Just as the original algorithm, its implementation guarantees the delivery of a message in less than a cycle's duration, and the detection of a node's failure within at most two cycle durations. In regard to entering the membership, and therefore being able to broadcast messages, a node can become a member within three cycle durations, although only under the condition that no other node is concurrently also attempting to join and no failures occur. Any of the latter conditions would delay the node's entrance by at least a cycle duration. In the previously mentioned case that there are only two members and one does not recognize a third node trying to enter, this node's entrance is delayed until the members decide to shut down due to the membership group being too small for a certain number of cycles.

### 7.1.3 Link Utilization

Regardless of the number of nodes involved, our schedule design causes one third of a cycle to be reserved for broadcasting. This means that, should a node not require to broadcast a message when it is its turn, or even be completely down, close to one third of the available bandwidth going out from said node is wasted. In order to improve throughput, it must be ensured that, if a node does go down, it is back up and running as quickly as possible. Furthermore, ideally the cycle duration allows for multiple messages to be collected, such that as much data as possible is transmitted during a node's sending slot.

## 7.2 Benchmark

### 7.2.1 Platform and Environment

The prototype was developed for a Supermicro SYS-E200-9B machine with the following equipment: four Intel Pentium N3710 1.60 GHz CPUs, 8 GiB of main memory, a Samsung SSD 860 EVO 500 GiB and four TSN capable Intel i210 NICs. The NICs are interconnected via a TL-SG3210 switch. The target device runs an Arch mainline Linux (kernel version 5.14.12). The *iperf3*[1] tool was used to determine a possible throughput of 955 Mbit/s between each pair of interfaces. To better simulate a distributed system, each node running the protocol is assigned to a different NIC and runs in a separate network namespace. In chapter 6, we noted that the communication between nodes requires a TSN capable switch that ensures that the protocol frames are not lost or delayed on their way from one node to the other. We unfortunately do not have such hardware at our disposal. However, our traffic only needs to travel through one switch before being broadcast back to the device's interfaces. Being an IEEE 802.1Q capable managed switch, the TL-SG3210 is set to assign one queue exclusively to VLAN frames where the PCP field is set to 5, i.e. our broadcast's priority.

When benchmarking features related to the communication aspect of the protocol, three nodes are run concurrently, i.e. the smallest number of nodes for which the protocol can still function. To better assess the prototypes' results, we run the benchmarks that we will present in the following for both the best-effort and TSN modes.

### 7.2.2 Timing Failures

In chapter 5, we defined as timing failures both an untimely computation resulting in one node becoming unsynchronized and a message arriving at a destination outside its designated window. In this section, we discuss the former type of timing failure, as it is the most common failure altogether in the context of our prototype. When such a failure is detected, it must be treated as a crash failure, i.e. the node shuts down immediately. Due to mainline Linux guaranteeing no determinism on the duration of kernel side operations, these type of failures are difficult to predict

[1] `https://iperf.fr/iperf-doc.php`, accessed December 12th 2021

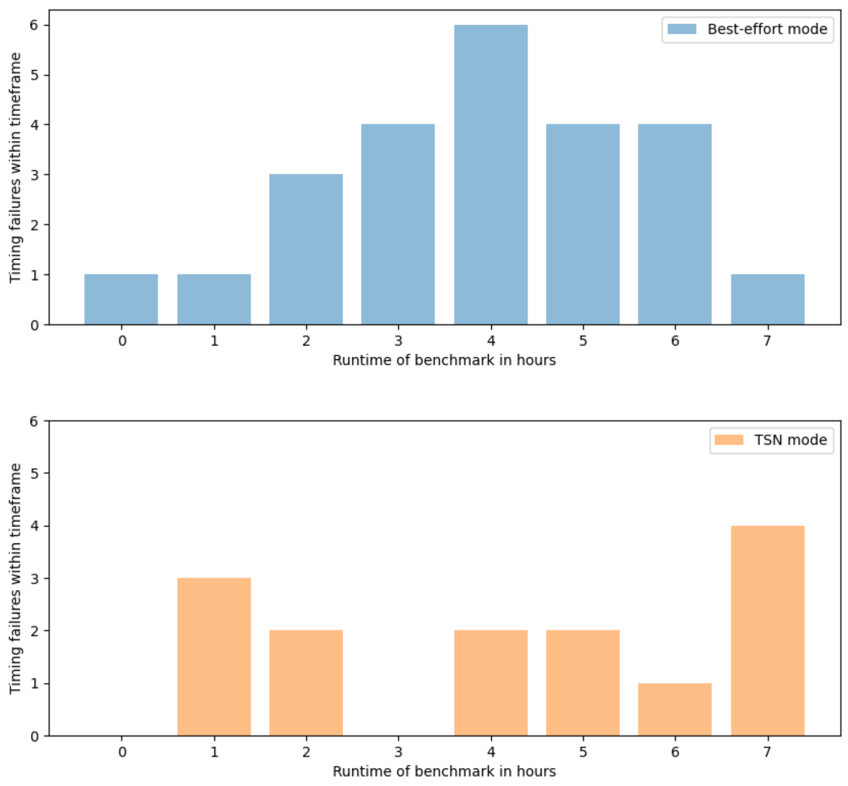

Figure 7.1: Timing failures observed over a timespan of 8 hours

and are very much platform dependent. Of course, a shorter slice duration, or computational window, increases the probability thereof. We attempted to evaluate how affected our prototype is by such scheduling delays based on the chosen slice duration. To this end, we ran an independent node on our target platform while reducing concurrent process activity to a bare minimum and setting all CPUs to the maximum frequency.

Throughout our experiments, we encountered great variance in scheduling delays causing timing failures over longer periods of time. Our benchmarks for a slice duration of 400 microseconds (fig. 7.1) show that, for both modes, different results can be achieved depending on the timespan in which the failures are observed. As a consequence, in order to get reliable results, a MTBF benchmark would have to be run over a longer timespan for each slice duration that is of interest.

Nevertheless, with respect to the prior considerations, we provide results for the MTBF across 20 runs for different slice durations (fig. 7.2). In our benchmark, the prototype has shown that with a slice duration of 200 microseconds a failure can be expected every few seconds, while for a slice duration of 300 microseconds the TSN mode is expected to run for approximately 46 seconds before failing. Increasing the slice duration to 400 microseconds, however, brings drastic improvement to the performance of the prototype, as the TSN mode is expected to run for over half an hour without failures. Finally, we find that timing failures are minimized for a slice duration greater than 500 microseconds. In order to achieve a deterministic protocol utilizing the target devices, one would have to choose a cycle duration that allows for slice durations larger than that value or otherwise run the protocol accepting

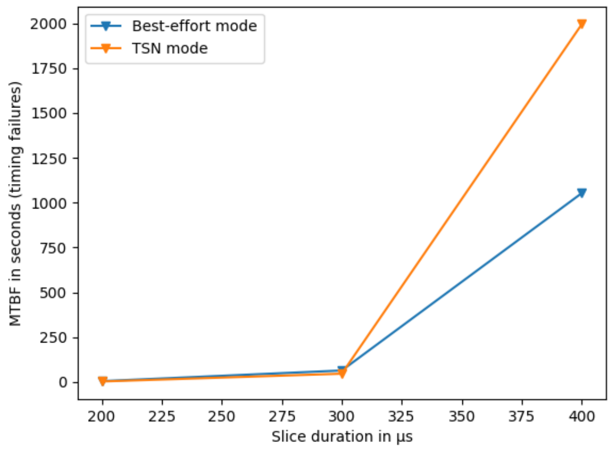

Figure 7.2: Mean time between timing failures

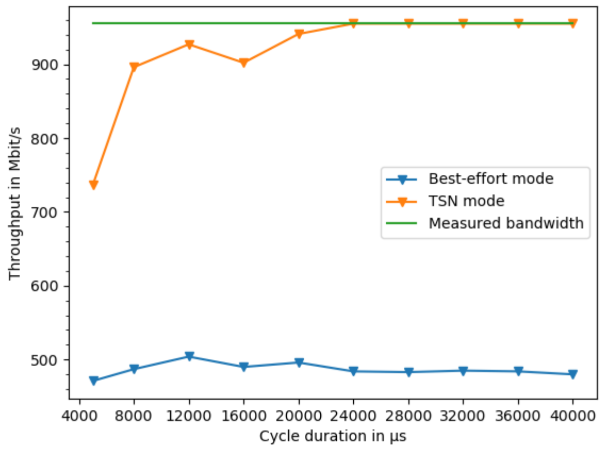

Figure 7.3: Best-effort throughput

occasional failures.

Although our results do not show a higher tendency towards timing failures by part of the TSN mode across all slice durations, we observe[2] that the system is affected by on average greater scheduling delays while using the *taprio qdisc* rather than the best-effort *multiqueue qdisc*.

### 7.2.3 Throughput

In section 7.1.3, we determined how much bandwidth an inactive node would waste in theory. In order to observe how the prototype performs in practice, we used the iperf3 tool to measure throughput in both the best-effort and TSN mode (fig. 7.3). The best-effort mode, as expected, only consumes as much bandwidth as it needs and allows for optimal throughput for cycle durations greater than 20 milliseconds.

[2]benchmarked with the *cyclictest* tool: `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start`, accessed December 12th 2021

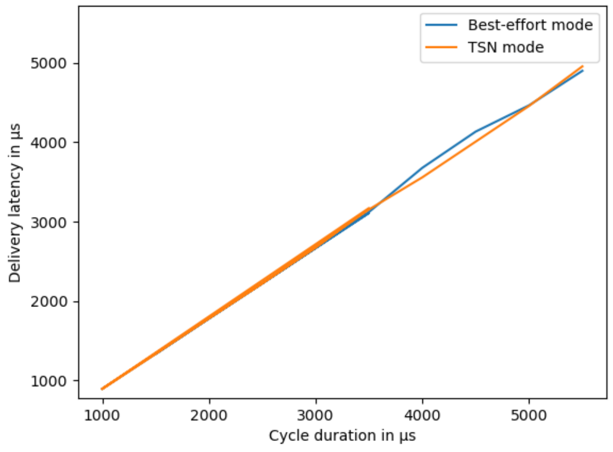

Figure 7.4: Message delivery latency

It is not fully clear why the measured throughput is worse for lower cycle durations, although it may be related to the fact that the iperf3 tool itself experiences scheduling delays due to the three nodes running with real-time priority. The TSN mode achieves consistent throughput, although it is around 50 % of the bandwidth rather than the expected two thirds discussed in section 7.1.3. This deviation, however, is most likely again caused by the measuring tool.

### 7.2.4 Latency Benchmark

In order to confirm the assumption that a broadcast message is always delivered within a cycle's duration, we tested both modes for different cycle durations. To this end, we took a timestamp whenever a message was broadcast and another when it was delivered to calculate the difference. Figure 7.4 shows that the prototype performs exactly as expected for both modes.

First, we run the benchmark with no additional load on the network. The results (fig. 7.5) very much favor the best-effort approach. The TSN approach not only achieves worse latencies, which can again be explained through the higher OS scheduling delays, but also higher jitter. A possible explanation would be that the large broadcasting window combined with no present load causes the *taprio* implementation to delay the transmission of some frames to a later point still within the same slice.

During the remaining benchmarks, we introduce background UDP traffic generated with the iperf3 tool. Moderate load on the network yields similar results as having no load at all, with best-effort clearly outperforming the TSN approach[3]. When generating a higher load of 800 Mbit/s, i.e. approximately 85% of the bandwidth, the limits of best-effort start to show (fig. 7.6). While many messages still arrive quickly, the number of messages experiencing higher delays increases. The performance of the TSN implementation, on the other hand, even performs better

[3]For both modes, introducing load has improved the latency by approximately 100 microseconds. However, this curious behavior has been also observed on other devices within the LAN using the ping tool, and is therefore not to be accredited to the prototype.

than with no load, as the vast majority of messages are broadcast earlier. We could not find the cause for this feature which is likely due to the *taprio* implementation.

If more traffic is generated than the receiver side can handle (therefore causing frames to be dropped), the best-effort mode cannot be run reliably at all, experiencing frequent and early receiver omission failures. Figure 7.7 shows that also for these kinds of loads, the TSN mode's approach is not affected. The best-effort mode, however, does not appear in the plot, since its frequent failures make it impossible to collect anywhere close to 6000 latencies. It can therefore be observed that, with such great load on the network, the best-effort approach is not viable.

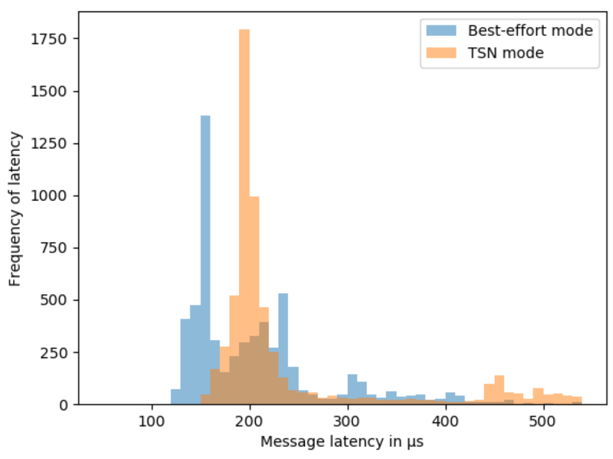

Figure 7.5: Message latency distribution with no load

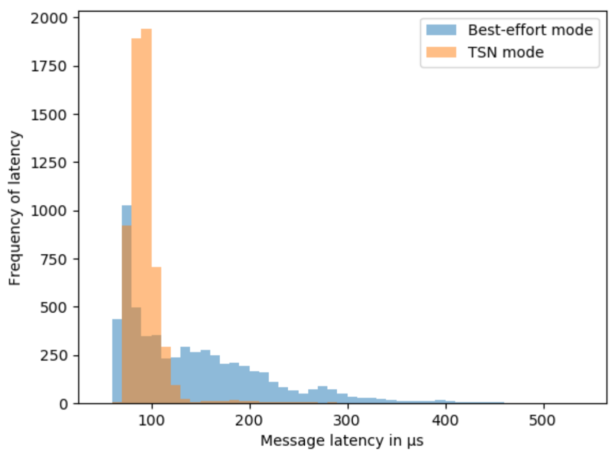

Figure 7.6: Message latency distribution with 85% load

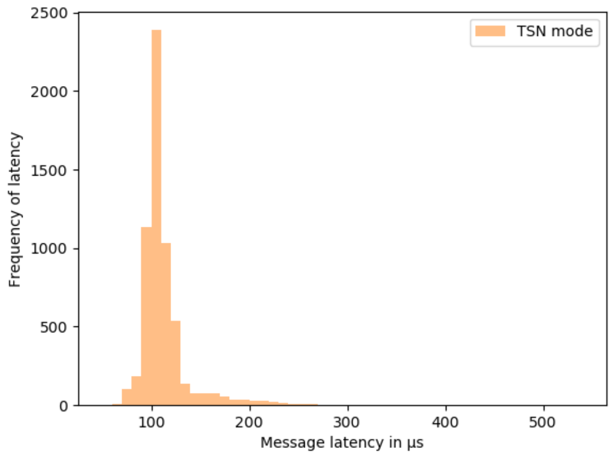

Figure 7.7: Message latency with maximum load

# Chapter 8

# Discussion and Outlook

Throughout this thesis we have showcased several benefits and shortcomings of both the algorithm and its TSN supported implementation. In this chapter, we discuss the prototype in regard to how it would apply to a (synchronous) data center. We also name ideas on how to improve the approach and mitigate the most critical weaknesses while providing an outlook on the synchronous data center and further applications of the TSN standards in a data center setting.

The first point we wish to discuss is the protocol header. In a data center setting where hundreds of thousands of packets travel through the links each day, the packet space dedicated to application level headers must be minimized. Although our implementation has led to a minimal increase in size, altogether, few bytes suffice to communicate the state of a member to the other participating nodes. Despite the positive results in this regard, further optimizations are possible to reduce the header size. For instance, transmitting the node identifier may be superfluous, as any identifier of a message's sender—such as a MAC address, already contained in the Ethernet header—can be mapped to a node identifier. Furthermore, information such as the current stage of a node and the suspicion information can be relayed in less than the three bytes the current implementations makes use of.

Additionally, the fact that the service provides both membership and Total Order Broadcast through one protocol can be used to also detect failures in a data center and, as already discussed in chapter 5, the bounded delivery latency works in its favor too. Unfortunately, this delivery latency strongly depends on the slice duration. The results from section 7.2.2 imply that by choosing a slice duration that minimizes timing failures, e.g. more than 500 microseconds on our target platform, a protocol involving 8 nodes would result in a delivery latency of no less than 12 milliseconds. Such a latency is nowhere close to being competitive with state of the art best-effort protocols [41]. Should a faster delivery latency be required, a lower slice duration must be picked in spite of introducing guaranteed failures. As even for slice durations with the most frequent failures a large amount of cycles can be completed, this solution can lead to more message delivery throughput, although problems arise when multiple failures delay a node (re)entrance indeterminately. Generally, a worse performance than best-effort solutions must be taken into consideration, since the goal of the approach is not to obtain the best performance but to achieve consistent performance that contributes to eliminating tail latency.

Another shortcoming of the approach that is partly related to timing failures is the low throughput evaluated in section 7.1.3 and section 7.2.3. Longer slice durations result in bigger sending slots (and therefore longer bandwidth reservations), while ideally the sending slot should be constrained by the latency and jitter caused by TSN communication. Regardless of the duration of a reservation, in order to make the most efficient use of it, a node must use the window to transmit as much data as possible by "packing" messages together [42]. However, in case throughout one cycle more messages are passed onto the service than can be broadcast in the upcoming sending slot, the broadcasting of some messages would have to be delayed by a further cycle. The time passing between the broadcast and the message delivery would still be deterministic, although the delay between the moments in time an application passes a message on to the service and the application gets confirmation that agreement was found would not be. In our implementation, we chose to omit such considerations altogether by making use of a trivial interface (section 6.2.3), but they are crucial in the effort of achieving predictable response times to client requests.

Many shortcomings presented by the implementation could be overcome by providing a bound on scheduling delays, which may delay important computations for an arbitrary amount of time. With a bound on such delays, one could profile node computations and communication precisely. This would allow for a separate assignment of appropriate and fixed computational windows that are independent from the cycle duration. Within the cycle one could then introduce variable "pause slots" that would function as a tradeoff between delivery latency and less frequent bandwidth reservations. Furthermore, in our small setting the prototype could not benefit fully from the bounded latency provided by TSN, as communication latencies are much shorter than the slice durations which allowed us to run the benchmarks. Bounded scheduling delays would also allow for a better assessment of TSN's performance.

The requirement of bounded OS scheduling delays cannot be fulfilled by a commodity OS as the one we chose for our solution. While the latter would allow for an easier integration into a data center, such a compromise makes the solution impracticable. Overall, this is the toughest challenge faced by the idea of a synchronous data center. A switch to synchrony implies deploying applications with real-time guarantees, which can be harder to develop and maintain. There would be less flexibility in resource allocation (e.g. static memory allocation for processes, i.e. no malloc). Both process and communication scheduling would have to be determined in a centralized, coordinated fashion. Choosing an appropriate schedule for one task may have appeared simple in our implementation, however doing the same for hundreds of applications spread across thousands of machines is definitely not a trivial task. As Yang et al. [3] point out, there is a fundamental lack of research when it comes to real-time scheduling in a data center setting. Further technologies from the CPS domain could help overcome such challenges. Adopting the *PREEMPT RT* Linux kernel patch[1] could prove to be a good compromise between using a commodity OS and having real-time guarantees, as the aim of the patch is to reduce the amount of kernel code that cannot be interrupted. Additionally, the IEEE 802.1Qcc standard could aid in the configuration of the TSN network schedule [43].

Other TDMA membership algorithms [44–46] may be transformed into a Total Order Broadcast with the same principles that were applied to the MARS member-

[1]https://wiki.linuxfoundation.org/realtime/start, accessed December 8th 2021

ship service in this thesis. Rosset et al.'s algorithm [45] is of particular interest, as it functions through dynamic slot assignment, meaning that—unlike in the MARS protocol—a node will not reserve bandwidth if it has no message to broadcast. While this solution may lead to less bandwidth being wasted, a TSN implementation appears difficult at the moment. *Taprio* schedules are static and, from our perspective, updating them is too time intensive for such an approach. Further, independently from the idea of a synchronous data center, plenty asynchronous Total Order Broadcast algorithms exist that rely on time synchronization [19]. The TSN IEEE 802.1AS standard alone could be applied to such an algorithm to achieve a more precise synchronization and thereby a better performance.

Finally, TSN technologies could allow the idea of a synchronous data center to be extended to inter-data center communication. The IETF DetNet Task Group[2] concerns itself with expanding the standards to Layer 3 [47], potentially allowing multiple networks to communicate in a deterministic fashion over the internet.

[2]`https://datatracker.ietf.org/wg/detnet/about/`, accessed December 8th 2021

# Chapter 9

# Conclusion

In this thesis, we observed the negative effects the best-effort service model can have on a data center environment and argued for a solution centered on synchronous communication. To this end we introduced TSN, a technology that we believe could soon be applied successfully to such environments in order to improve latencies. Furthermore, we discussed Total Order Broadcast, its importance in regard to data centers and presented a TDMA Total Order Broadcast algorithm [26]. We observed that the algorithm fits well to the idea of a synchronous data center [3], although we also pointed out its shortcomings in terms of throughput.

In addition, we presented our implementation of the algorithm, and showed how we integrate the Time-Aware Shaper to achieve deterministic communication with relative ease. The results of the evaluation show that the prototype maintains the benefits of the algorithm, i.e. a bounded message delivery and low protocol overhead, but also encounters multiple problems related to OS scheduling delays which turn out to be a bigger bottleneck than the communication. Moreover, our benchmarks could measure a slightly lower throughput compared to what we determined in a static analysis. However, it can be observed that, confronted with a best-effort version of the protocol, the TSN based solution performs better in presence of severe network congestion. In our discussion we then outlined the implications of problems such as the aforementioned scheduling delays, and argued in favor of making use of an OS that provides real-time guarantees. We questioned whether the results one can potentially achieve through synchronous computation are worth the effort that comes with the development of real-time software.

Nevertheless, our prototype has shown how TSN can achieve significant results outside of its comfort zone. The Linux implementation of the standards is currently still in progress and other TSN standards like IEEE 802.1Qcc would simplify the integration of a synchronous schedule into the network. The path towards a synchronous data center is still long and tedious, but in this thesis TSN has shown that it can contribute to it.

# Bibliography

[1] T. N. Theis and H.-S. P. Wong, “The end of Moore’s law: A new beginning for information technology,” *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[2] R. Chen and G. Sun, “A survey of kernel-bypass techniques in network stack,” in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, 2018, pp. 474–477.

[3] T. Yang, R. Gifford, A. Haeberlen, and L. T. X. Phan, “The synchronous data center,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 142–148.

[4] D. Clark, “The design philosophy of the DARPA internet protocols,” in *Symposium proceedings on Communications architectures and protocols*, 1988, pp. 106–114.

[5] H. Kopetz, *Real-time systems: design principles for distributed embedded applications.* Springer Science & Business Media, 2011.

[6] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, “Fastpass: A centralized “zero-queue” datacenter network,” in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 307–318.

[7] S. Ryu, C. Rump, and C. Qiao, “Advances in internet congestion control,” *IEEE Communications Surveys & Tutorials*, vol. 5, no. 1, pp. 28–39, 2003.

[8] S. Shenker, “Fundamental design issues for the future internet,” *IEEE Journal on selected areas in communications*, vol. 13, no. 7, pp. 1176–1188, 1995.

[9] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, “Towards predictable datacenter networks,” in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, pp. 242–253.

[10] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, “Tales of the tail: Hardware, OS, and application-level sources of tail latency,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[11] A. Althoubi, R. Alshahrani, and H. Peyravi, “Tail latency in datacenter networks,” in *Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2020, pp. 254–272.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.

[13] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2017.

[14] E. Arjomandi, M. J. Fischer, and N. A. Lynch, "A difference in efficiency between synchronous and asynchronous systems," in *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, 1981, pp. 128–132.

[15] N. Santoro, *Design and analysis of distributed algorithms.* John Wiley & Sons, 2006.

[16] A. Galleni and D. Powell, "Consensus and membership in synchronous and asynchronous distributed systems," University of Bologna, Tech. Rep., 1995.

[17] S. Frølund and F. Pedone, "Dealing efficiently with data-center disasters," *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1064–1081, 2003.

[18] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *Proceedings of the 15th International Middleware Conference*, 2014, pp. 169–180.

[19] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[20] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Tech. Rep., 1994.

[21] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[22] M. Van Steen and A. Tanenbaum, *Distributed systems principles and paradigms*, 2002.

[23] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, no. 2, pp. 95–114, 1978.

[24] J. N. Gray, "Notes on data base operating systems," in *Operating Systems.* Springer, 1978, pp. 393–481.

[25] O. Babaoglu and S. Toueg, "Understanding non-blocking atomic commitment," University of Bologna, Tech. Rep., 1993.

[26] H. Kopetz, G. Grünsteidl, and J. Reisinger, "Fault-tolerant membership service in a synchronous distributed real-time system," in *Dependable Computing for Critical Applications.* Springer, 1991, pp. 411–429.

[27] M. H. Farzaneh and A. Knoll, "Time-sensitive networking (TSN): An experimental setup," in *2017 IEEE Vehicular Networking Conference (VNC)*, 2017, pp. 23–26.

[28] J. L. Messenger, "Time-Sensitive Networking: An introduction," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 29–33, 2018.

[29] A. S. Tanenbaum, D. Wetherall *et al.*, *Computer networks.* Prentice hall, 1996.

[30] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012, pp. 1–14.

[31] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," *Information and Computation*, vol. 118, no. 1, pp. 158–179, 1995.

[32] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 77–97, 1987.

[33] I. Stanoi, D. Agrawal, and A. El Abbadi, "Using broadcast primitives in replicated databases," in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No. 98CB36183)*, 1998, pp. 148–155.

[34] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 551–566, 2005.

[35] H. Kopetz and G. Grunsteidl, "TTP-A time-triggered protocol for fault-tolerant real-time systems," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993, pp. 524–533.

[36] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The Mars approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, 1989.

[37] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The time-triggered ethernet (TTE) design," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, 2005, pp. 22–33.

[38] N. Finn, "Introduction to Time-Sensitive Networking," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 22–28, 2018.

[39] W. Steiner, S. S. Craciunas, and R. S. Oliver, "Traffic planning for time-sensitive communication," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 42–47, 2018.

[40] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[41] G. Berthou and V. Quéma, "Fastcast: a throughput-and latency-efficient total order broadcast protocol," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2013, pp. 1–20.

[42] R. Friedman and R. Van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)*, 1997, pp. 233–242.

[43] S. Brooks and E. Uludag, "Time-Sensitive Networking: From theory to implementation in industrial automation," *Intel white paper*, 2018.

[44] P. D. Ezhilchelvan and R. de Lemos, "A robust group membership algorithm for distributed real-time systems," in *[1990] Proceedings 11th Real-Time Systems Symposium*, 1990, pp. 173–179.

[45] V. Rosset, P. F. Souto, and F. Vasques, "A group membership protocol for communication systems with both static and dynamic scheduling," in *IEEE International Workshop on Factory Communication Systems-Proceedings, WFCS*, 2006.

[46] R. Barbosa, A. Ferreira, and J. Karlsson, "Implementation of a flexible membership protocol on a real-time ethernet prototype," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, 2007, pp. 342–347.

[47] X. Yang, D. Scholz, and M. Helm, "Deterministic networking (DetNet) vs Time-Sensitive Networking (TSN)," in *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM)*, 2019, pp. 79–84.