Algoritmo Multicast Generalizado: Formalização e Validação

José Augusto Bolina

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2022

José Augusto Bolina

Algoritmo Multicast Generalizado: Formalização e Validação

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Supervisor: Lásaro Camargos
Co-supervisor: Rafael Pasquini

Uberlândia
2022

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

L933 2022

Lucas, José Augusto Bolina, 1997-
Algoritmo Multicast Generalizado [recurso eletrônico] : Formalização e Validação / José Augusto Bolina Lucas. - 2022.

Orientador: Lásaro Jonas Camargos.
Coorientador: Rafael Pasquini.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2022.596
Inclui bibliografia.

1. Computação. I. Camargos, Lásaro Jonas,1978-, (Orient.). II. Pasquini, Rafael,1981-, (Coorient.). III. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. IV. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação de Mestrado 15/2022, PPGCO				
Data:	29 de setembro de 2022	Hora de início:	08:30	Hora de encerramento:	10:15
Matrícula do Discente:	12022CCP004				
Nome do Discente:	José Augusto Bolina Lucas				
Título do Trabalho:	Algoritmo Multicast Generalizado: Formalização e Validação				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Paulo Rodolfo da Silva Leite Coelho - FACOM/UFU; Fernando Lopes Pedone - USI; Rafael Pasquini - FACOM/UFU (coorientador) e Lásaro Jonas Camargos, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Fernando Pedone - Lugano/Suiça; Paulo Rodolfo da Silva Leite Coelho, Rafael Pasquini e Lásaro Jonas Camargos - Uberlândia/MG. O discente participou da cidade de Uberlândia/MG.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Lásaro Jonas Camargos, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Lasaro Jonas Camargos**, **Usuário Externo**, em 24/10/2022, às 10:25, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Fernando Pedone**, **Usuário Externo**, em 24/10/2022, às 15:08, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Rafael Pasquini**, **Professor(a) do Magistério Superior**, em 24/10/2022, às 15:27, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Paulo Rodolfo da Silva Leite Coelho**, **Professor(a) do Magistério Superior**, em 24/10/2022, às 15:27, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3957177** e o código CRC **A565AE0D**.

Acknowledgements

Gostaria de agradecer toda a minha família, amigos e meus orientadores Lásaro Camargos e Rafael Pasquini pela ajuda, dedicação e oportunidade.

Um grande agradecimento aos meus amigos pela ajuda nos estudos e pelas noites de descontração. Um agradecimento especial para minha companheira Laura Ribeiro por toda paciência, compreensão, e ajuda nesse período.

"É verdade
sem mentira
certo muito
verdadeiro"
(Jorge Ben)

Resumo

Algoritmos de sistemas distribuídos são peças essenciais para criação de aplicações tolerante a faltas. A corretude desses algoritmos é crucial. Nesse sentido, o presente trabalho formaliza e especifica três algoritmos para multi-difusão generalizada utilizando TLA$^+$, corrigindo os problemas encontrados durante o processo. Em um lado mais prático, implementamos um protótipo de um dos algoritmos corrigidos. O presente trabalho detalha os algoritmos, os problemas encontrados e as respectivas soluções, e finalmente, o processo de especificação e implementação.

Palavras-chave: Consenso; Tolerância a faltas; Difusão Genérica; Difusão Atômica.

Abstract

Distributed systems algorithms are an essential building block to creating fault-tolerant applications. The correctness of such algorithms is crucial. The current work formalizes and specifies three generic multicast algorithms using TLA$^+$. We detail the formalization process, describing the problems and their corrections. On a more practical side, we implement a prototype of one of the specified algorithms. The current work aims to describe the process of (i) formalization and correction of three generic multicast algorithms and (ii) implementation of an algorithm directly from the specification.

Keywords: Consensus; Fault-Tolerance; Generic Multicast; Atomic Multicast.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Computer systems are ubiquitous in day-to-day life, with different kinds of applications, where the critical ones must have high availability and correctly behave when requested. Distributed applications can offer high availability and fault tolerance by using group communication primitives that offers varying properties, selecting the most adequate depending on the application's requirements.

There are different flavors of group communication primitives, each with its own guarantees and requirements. For example, Reliable Broadcast can reliably deliver messages to all participating processes; other primitives enforce an order to the message delivery, such as FIFO, causal, and total (DÉFAGO; SCHIPER; URBÁN, 2000). Variants abound, with a corresponding variety of algorithms implementing them (PEDONE; SCHIPER, 1999; LAMPORT et al., 2001; ONGARO; OUSTERHOUT, 2014).

The multicast family offers more flexible primitives when compared to the broadcast family. The primitive known as Atomic Multicast can reliably deliver a message in the same order to a subset of processes in the system. The ordering and reliable delivery guarantees make the Atomic Multicast primitive interesting for implementing the state machine replication technique, commonly used to implement fault-tolerant services in distributed systems (SCHNEIDER, 1990). Informally explaining, by creating a set of replicas of a deterministic process, starting all of them in the same state, applying the same sequence of commands, everyone proceeds the same (LAMPORT, 1994b). Even if some of the replicas fail, there are others to provide the service.

A more generalized approach could order messages only when required since not every pair of operations needs total order; partial ordering commands may be enough. A Generic Multicast algorithm can create a partial order of messages, having a generalized behavior. If message ordering adds a cost to the algorithm, a less expensive algorithm avoids it (PEDONE; SCHIPER, 1999).

ANTUNES proposed three new algorithms that solve the Generic Multicast problem in an unpublished work. The proposed algorithms extend previous work in the literature, adding the aforementioned generalized behavior. First, changing the Atomic Multicast algorithm proposed by Skeen that works in a failure-free environment (BIRMAN; JOSEPH, 1987), the proposed algorithm is just an introduction, and it is called Generic Multicast 0. Then, extending FRITZKE et al.'s algorithm with improvements proposed by SCHIPER; PEDONE, called Generic Multicast 1. The final algorithm is called Generic Multicast 2, created from the previous proposals by replacing the Atomic Broadcast with a Generic Broadcast primitive.

Like many other algorithms proposed before, ANTUNES presents its algorithms in pseudo-code, but those are not formally verified. While this has been a long-standing practice, formal methods are considered very expensive. Recent developments have pushed for better specification and verification of algorithms, for example, applying formal methods to check the correctness of developed artifacts (BORNHOLT et al., 2021) and embedding specification of the problem and solution during the development process (SYSTEMS, 2020). Such a formalization gives higher confidence in the algorithm's correctness.

1.1 Contributions

In this work, we have formally specified the algorithms proposed by ANTUNES using TLA$^+$ and checked the specifications using TLC. In this effort, we have identified several problems which we have rectified. During the process, we came up with a version that uses fewer communication primitives and removes one intra-group message exchange. We implemented a prototype for the newly proposed algorithm using the Go language (GOLANG, 2021b). Both the prototype[1] and the TLA$^+$ specifications[2] are available for public scrutiny.

1.2 Organization

The current work starts by laying a theoretical foundation in Chapter 2. First, we will introduce the system model, communication primitives, and notation used throughout this work. The chapter explains what TLA is and how a system is formally specified using TLA$^+$. The chapter finishes with a quick overview of Go, the programming language used to develop the Generic Multicast 1 prototype.

Chapter 3 is a discussion of related works. We discuss the genealogy of the algorithms developed by ANTUNES, which forms the basis of our algorithms. We also briefly discuss other works focused on formally specifying algorithms and others related to consensus algorithms implementation.

Chapters 4 and Chapter 5 are the meat of the current work. Initially, the correctness verification from the previously proposed algorithms, a story told in detail, describing the

[1] <https://github.com/jabolina/go-mcast>
[2] <https://github.com/jabolina/mcast-tlaplus>

complete process of writing the specification, the problems found, and a final corrected proposition. We extend the discussion with additional properties of the algorithms and establish behavior propositions with proofs left for future work. The other chapter is the prototyping process, describing the modeling of data structures from the specification into a programming language, the tests, and the required communication primitives. Although the chapters are separated, we did some work in parallel.

Chapter 6 concludes the current work. This chapter contains a summary and lists potential future work.

Chapter 2

Fundamentals

This chapter reviews the concepts used throughout this work. Section 2.1 defines the processes and the primary piece of communication; Section 2.2 presents the Consensus problem. Having processes and communication channels, we may want to create fault-tolerant applications to keep working even if some parts of the system have failed, so in Section 2.3.1, we discuss the state machine replication technique. With the introduction of groups for replication, Section 2.3.2 will discuss its definitions, properties, and communication primitives.

The remaining sections in this chapter present the tools we use to specify and implement the algorithms. Section 2.4 discusses what we use for specification and formalization, TLA, TLA$^+$, and TLC. And Section 2.5, for implementation, the programming language Golang.

2.1 Processes and Channels

The algorithms work in an environment, making assumptions and defining requirements. Each of our proposed algorithms works in its specific environment. Here we establish definitions for all of our algorithms and increment them step-by-step in the following sections.

2.1.1 The Universe and Everything Else

The systems are composed of processes and communication channels. The set of all processes is $\Pi = \{p_1, p_2, ..., p_n\}$, where they share neither memory nor a global clock and communicate only by message-passing through the communication channels. The communication channels connect every pair of processes and provide two basic primitives to send and receive messages. A *message* is a tuple of values. For a process $p_i \in \Pi$ to send a tuple t to a process $p_j \in \Pi$, p_i invokes $\boxed{\text{Send } t \text{ to } p_j}$, and when the target receives the tuple, $\boxed{\text{Received } t \text{ from } p_i}$ is invoked at p_j.

Usually, tuples are constructed in place when sending and are pattern-matched when received. For example, to send a message m and a timestamp ts to a process p_j, p_i would invoke $\boxed{\text{Send } \langle m, ts \rangle \text{ to } p_j}$ and the reception as $\boxed{\text{Received } \langle m, ts \rangle \text{ from } p_i}$. Pattern matching requires the tuples to match in size and uses _ to mean that any value matches the corresponding element in the position, which, in turn, is discarded. These primitives implement a *quasi-reliable* communication with the following properties (PEDONE; SCHIPER, 2002):

- **No creation**: for $p_i, p_j \in \Pi$, if $\boxed{\text{Received } t \text{ from } p_i}$ is invoked in p_j, then p_i must have invoked $\boxed{\text{Send } t \text{ to } p_j}$;
- **No duplication**: for $p_i, p_j \in \Pi$, for every $\boxed{\text{Send } t \text{ to } p_j}$ invoked by p_i, a corresponding $\boxed{\text{Received } t \text{ from } p_i}$ is invoked in p_j at most once;
- **No loss**: for $p_i, p_j \in \Pi$, if process p_i invokes $\boxed{\text{Send } t \text{ to } p_j}$, and if neither p_i nor p_j fails, then eventually $\boxed{\text{Received } t \text{ from } p_i}$ is invoked in p_j.

2.1.2 Failing

The last property states that if the sender or receiver fails, the message might be lost, but what does failing means? We define that if a process behaves exclusively according to its specification, it is correct. If it ceases working or deviates from the specification, it is incorrect; it fails. We do not consider malicious processes.

Our algorithms adopt a different failure model, so we make it explicit when presenting the algorithms. Common to all algorithms is that the system is asynchronous, without assumptions about process speed or message delivery time (ANTUNES, 2019).

2.2 The Consensus Problem

Informally, we can define the consensus problem as a collection of servers proposing values and eventually agreeing upon one of such proposals (DÉFAGO; SCHIPER; URBÁN, 2004). More formally, an algorithm that solves the consensus problem fulfills the following properties (CHANDRA; TOUEG, 1996):

- **Agreement**: no two correct processes $p_1, p_2 \in \Pi$ can agree on different values;
- **Integrity**: every correct process in Π agrees at most once;
- **Validity**: if a correct process $p_i \in \Pi$ agrees on a value v, then v was previously proposed by a correct process $p_j \in \Pi$;
- **Termination**: every correct process in Π eventually agrees on some value;

This consensus specification allows different agreed values if one of the processes is incorrect (CHARRON-BOST; SCHIPER, 2004). The *uniform* consensus variant derives the consensus properties, dealing with incorrect processes. *Correctness* properties, like Agreement, Integrity, and Validity, must hold irrespective of whether the process is correct or incorrect. *Liveness* properties, like Termination, are harder to enforce in a misbehaving process, so the uniform variant of consensus only requires Termination to correct ones (FRITZKE et al., 1998; DÉFAGO; SCHIPER; URBÁN, 2004). The properties are then (CHARRON-BOST; SCHIPER, 2004; CHANDRA; TOUEG, 1996):

- **Uniform Agreement**: no two processes $p_1, p_2 \in \Pi$ can agree on different values;
- **Uniform Integrity**: every process in Π agrees at most once;
- **Uniform Validity**: if a process p in Π agrees on a value v, then v was previously proposed by a process q in Π;
- **Termination**: every correct process in Π eventually agrees on some value;

This consensus definition is impossible to solve in an asynchronous system if even a single process is incorrect, a result known as the *FLP impossibility* (FISCHER; LYNCH; PATERSON, 1985). The impossibility arises from the fact that it is not possible to guarantee Termination (FISCHER; LYNCH; PATERSON, 1985); a slow process is indistinguishable from an incorrect one on an asynchronous system. One solution to allow solving the consensus in asynchronous systems with failures is augmenting the system with a mechanism known as a *failure detector* (CHANDRA; TOUEG, 1996).

We define the set of all failure detectors as $\mathcal{D} = \{d_1,\ d_2,\ ...,\ d_n\}$ (CHANDRA; TOUEG, 1996). Each process $p_i \in \Pi$ has an attached local failure detector module d_i. When $p_i \in \Pi$ queries its local failure detector d_i, the response can be *incorrect* by incorrectly suspecting a process; and can be *inconsistent* when at time t, detector d_j suspects a process p_n and detector d_i does not (DÉFAGO; SCHIPER; URBÁN, 2000). These are the properties of *completeness* and *accuracy* (DÉFAGO; SCHIPER; URBÁN, 2000), respectively.

The work of CHANDRA; TOUEG shows that the weakest failure detector needed to solve the consensus problem in an asynchronous system is the $\diamond\mathcal{W}$, equivalent to $\diamond\mathcal{S}$ (CHANDRA; HADZILACOS; TOUEG, 1996; CHANDRA; TOUEG, 1996; DÉFAGO; SCHIPER; URBÁN, 2004). This failure detector has the following properties:

- **Strong completeness**: eventually, every correct process permanently suspects a process that failed;
- **Eventual weak accuracy**: eventually, a correct process is not suspected by any correct process.

But how can we glue this together to create a fault-tolerant application?

2.3 Fault Tolerance and Groups

Some applications are critical, and therefore, such applications must be able to tolerate system faults. We can replicate an application amongst the available multiple machines, thus providing redundancy, high availability, and fault tolerance, where if one fails, others can continue working (KLEPPMANN, 2017; CHARRON-BOST; PEDONE; SCHIPER, 2010). The hardship of replication is dealing with stateful applications where the data changes (KLEPPMANN, 2017). Here we will discuss the state-machine replication technique.

2.3.1 State Machine Replication

State Machine Replication is a technique to create fault-tolerant applications in distributed systems (SCHNEIDER, 1990). In this approach, a server is designed as a deterministic state machine and replicated on a collection of servers; by starting all servers with the same state, applying the same sequence of commands to the server replicas in the same order, the output is the same (SCHNEIDER, 1990). Even when some servers are unavailable, the system can still operate, thus being fault-tolerant. Sometimes *state machine*, *server*, and *replicas* are used interchangeably, but in this work, we only refer to them as servers.

Using a consensus algorithm, a collection of servers agree on a single value and can work as a consistent group (LAMPORT et al., 2001). If multiple consensus instances execute in sequence, such that the i^{th} consensus instance agrees on the i^{th} command, then all deterministic server proceed through the same states (LAMPORT et al., 2001).

2.3.2 Communication Primitives for Groups

With a replicated application, we are not dealing with a single process; we are dealing with a group of processes. In these scenarios, primitives for group communication are more desirable. These primitives are designed to handle groups and provide varying guarantees, working similarly to the primitives for process communication, transporting an abstract structure, which we say is a message for simplicity.

The group operation has groups as the destination, which are subsets of Π. In the context of this work, the set of groups is defined *a priori* as $\Gamma = \{g_1, g_2, ..., g_n\}$, and $g_i \subseteq \Pi$. In fact, we consider that groups neither are empty ($\forall g \in \Gamma, g \neq \emptyset$) nor overlap ($\forall g_i, g_j \in \Gamma, j \neq i : g_j \cap g_i = \emptyset$) and that all processes must belong to one group ($\forall p \in \Pi : \exists g \in \Gamma : p \in g$). We use *groups* and *partitions* as synonyms.

Groups are either *static*, if they cannot change throughout the algorithm's execution, or *dynamic*, otherwise. In this work, for simplicity, we consider static groups, although using group membership protocols and adaptations to the algorithms, it would be possible to use dynamic groups instead (SCHIPER, 2006).

Groups can also be *open* or *closed*. In a system with closed groups, a message sent to a group $g \in \Gamma$ requires the sender to also be in g, meaning that the sender process must be in the destination group (DÉFAGO; SCHIPER; URBÁN, 2000). However, an open group can receive messages from any process in Π, being more general and providing better support for distributed systems (DÉFAGO; SCHIPER; URBÁN, 2000). The algorithms presented here use open groups (ANTUNES, 2019).

With all the formalisms out of the way, we can start looking at the group primitives. Our algorithms build on top of these primitives.

2.3.2.1 Reliable Broadcast

An algorithm that solves the reliable broadcast problem provides a group communication to broadcast messages to one group with delivery guarantees. Guarantee that, for a correct sender, all correct processes in the addressed group eventually deliver the message. For an incorrect sender, or either every correct process or none delivers the message. (DÉFAGO; SCHIPER; URBÁN, 2004).

Formally, we define reliable broadcast through the primitives $\boxed{\text{rb-Send } m \text{ to } \mathcal{D}}$, used by a process $p \in \Pi$ to broadcast a message m to all processes in $\mathcal{D}$, where either $\mathcal{D} = \Pi$ or $\in \Gamma$; and $\boxed{\text{rb-Delivered } m}$, in which a process $p \in \mathcal{D}$ delivers a message m. These primitives satisfy the following properties:

- **Validity**: if a correct process in Π $\boxed{\text{rb-Send } m \text{ to } \mathcal{D}}$, then all correct processes in $\mathcal{D}$ eventually $\boxed{\text{rb-Delivered } m}$;
- **Agreement**: if a process in $\mathcal{D}$ $\boxed{\text{rb-Delivered } m}$, then all correct processes in $\mathcal{D}$ eventually $\boxed{\text{rb-Delivered } m}$;
- **Integrity**: for any message m, every process in $\mathcal{D}$ $\boxed{\text{rb-Delivered } m}$ at most once, and only if m was previously $\boxed{\text{rb-Send } m \text{ to } \mathcal{D}}$ by a process in Π.

2.3.2.2 Atomic Broadcast

In the atomic broadcast problem, also known as total order broadcast, a process can reliably send messages to all processes in the system, as in the reliable broadcast problem, while guaranteeing that all messages are delivered in the same order by all recipients (DÉFAGO; SCHIPER; URBÁN, 2000). The problem is defined in terms of primitives $\boxed{\text{ab-Send } m \text{ to } \mathcal{D}}$, used by a process in Π to broadcast a message m to all processes in $\mathcal{D}$, where either $\mathcal{D} = \Pi$ or $\in \Gamma$; and $\boxed{\text{ab-Delivered } m}$, in which a process $p \in \mathcal{D}$ delivers a message m. Formally, an atomic broadcast primitive satisfies the following properties (DÉFAGO; SCHIPER; URBÁN, 2000):

- **Validity**: if a correct process in Π $\boxed{\text{ab-Send } m \text{ to } \mathcal{D}}$, then all correct processes in $\mathcal{D}$ eventually $\boxed{\text{ab-Delivered } m}$;

- **Agreement**: if a process in $\mathcal{D}$ $\boxed{\text{ab-Delivered } m}$, then all correct processes in $\mathcal{D}$ eventually $\boxed{\text{ab-Delivered } m}$;

- **Integrity**: for any message m, every process in $\mathcal{D}$ $\boxed{\text{ab-Delivered } m}$ at most once, and only if m was previously $\boxed{\text{ab-Send } m \text{ to } \mathcal{D}}$ by a process in Π;

- **Total Order**: if processes $p_1, p_2 \in \mathcal{D}$ both delivers messages m_1 and m_2, then p_1 $\boxed{\text{ab-Delivered } m_1}$ before $\boxed{\text{ab-Delivered } m_2}$ if, and only if, p_2 $\boxed{\text{ab-Delivered } m_1}$ before $\boxed{\text{ab-Delivered } m_2}$.

The atomic broadcast primitive satisfies all the requirements for a reliable broadcast primitive, adding a more strict property for totally ordering all messages. Atomic broadcast is equivalent to the consensus problem described previously in Section 2.2 (DÉFAGO; SCHIPER; URBÁN, 2004). In fact, an infinite sequence of consensus instances can implement the atomic broadcast. The converse side of the equivalence is straightforward: to propose values for the consensus, just broadcast them; the value decided is the first delivered by the atomic broadcast protocol. One important implication of this equivalence is that the same failure detector needed to solve consensus is needed to solve atomic broadcast.

Any process in Π can use the primitive $\boxed{\text{ab-Send } m \text{ to } \mathcal{D}}$, but $\mathcal{D}$ must be a single group. In some situations, it may be necessary to have multiple groups in destination for the same messages with reliability and total order guarantees. In this case, the atomic multicast primitives are better adequate.

2.3.2.3 Atomic Multicast

The atomic multicast problem, also known as total order multicast, is defined in terms of a destination set $\mathcal{G} \subseteq \Gamma$, and the primitives $\boxed{\text{am-Send } m \text{ to } \mathcal{G}}$, used by a process in Π to multicast a message m to processes in $\bigcup_{\mathcal{G}}$; and the primitive $\boxed{\text{am-Delivered } m}$, which processes in $\bigcup_{\mathcal{G}}$ deliver a message m.

The properties that must be satisfied by an atomic multicast algorithm are similar to those of atomic broadcast algorithms, although not equal:

- **Validity**: if a correct process $p \in \Pi$ $\boxed{\text{am-Send } m \text{ to } \mathcal{G}}$, $\mathcal{G} \subseteq \Gamma$, then all correct processes in $\bigcup_{\mathcal{G}}$ eventually $\boxed{\text{am-Delivered } m}$;

- **Agreement**: if a process in $\bigcup_{\mathcal{G}}$, $\mathcal{G} \subseteq \Gamma$, $\boxed{\text{am-Delivered } m}$, then all correct processes in $\bigcup_{\mathcal{G}}$ eventually $\boxed{\text{am-Delivered } m}$;

- **Integrity**: for any message m and every process $p \in \bigcup_{\mathcal{G}}$ that $\boxed{\text{am-Delivered } m}$, where $\mathcal{G} \subseteq \Gamma$, p $\boxed{\text{am-Delivered } m}$ at most once and only if m was previously $\boxed{\text{am-Send } m \text{ to } \mathcal{G}}$ by some process in Π;
- **Total Order**: given two messages m_1 and m_2 and two processes p_i, $p_j \in \Pi$, if both p_1 and p_2 $\boxed{\text{am-Delivered } m_1}$ and $\boxed{\text{am-Delivered } m_2}$, then p_i $\boxed{\text{am-Delivered } m_1}$ before $\boxed{\text{am-Delivered } m_2}$ if, and only if p_2 $\boxed{\text{am-Delivered } m_1}$ before $\boxed{\text{am-Delivered } m_2}$.

An asynchronous system must have the $\diamond\mathcal{S}$ failure detector to solve the atomic multicast (and broadcast) problems (DÉFAGO; SCHIPER; URBÁN, 2000; DÉFAGO; SCHIPER; URBÁN, 2004). The atomic multicast primitive provides the same guarantees as the atomic broadcast, whereas, in fact, one may see the atomic broadcast problem as a specific case of atomic multicast with a single group in $\mathcal{G}$ (a single partition of Π). Atomic Multicast can solve Atomic Broadcast by sending messages to all participants (GUERRAOUI; SCHIPER, 1997; DÉFAGO; SCHIPER; URBÁN, 2004). The Atomic Broadcast can solve Atomic Multicast by broadcasting the tuple $\langle$message, destination$\rangle$, and the processes discard messages when it is not present in the destination. The second approach creates a feigned Atomic Multicast algorithm because it involves more members than necessary, creating an algorithm as costly as the broadcast (GUERRAOUI; SCHIPER, 1997). The following minimality property asserts that an algorithm is not feigned (GUERRAOUI; SCHIPER, 1997):

- **Minimality**: An algorithm that implements the Atomic Multicast of a message m to a destination $\mathcal{G}$ involves only the sender process and the processes in $\mathcal{G}$.

An algorithm that solves Atomic Multicast using Atomic Broadcast is not genuine (GUERRAOUI; SCHIPER, 1997). This property ensures that only necessary processes participate in message delivery.

2.3.2.4 Generic Broadcast

The atomic broadcast problem delivers messages in total order. The ordering guarantee, however, may be too strong for the application that is using it. A simple and concrete example is that of a distributed counter, where this counter receives operations for adding and multiplying its current value. Addition operations do not need to have a total order with other addition operations, and the same applies to multiplication. Although, when we mix these operations, we must have an ordering guarantee between addition and multiplication.

In such scenarios, a primitive with a generalized behavior fits better. The generic broadcast is one of these primitives, where it uses the messages' *semantic* information to determine whether messages need order and effectively deliver them in a partial order, different from the total order of atomic broadcast (PEDONE; SCHIPER, 1999; PEDONE;

SCHIPER, 2002; CAMARGOS, 2008). In the generic broadcast, a *conflict relation* captures the semantic information, specifying which pair of messages commute. We say that conflicting messages do not commute; if they do not conflict, they commute.

We define generic broadcast by the primitives $\boxed{\text{gb-Send } m \text{ to } \mathcal{D}}$, used by a process in Π to broadcast a message m to all processes in $\mathcal{D}$, where either $\mathcal{D} = \Pi$ or $\mathcal{D} \in \Gamma$; $\boxed{\text{gb-Delivered } m}$, in which a process in $\mathcal{D}$ delivers a message m; and the conflict relation, defined as $\mathcal{C}$, symmetric, non-reflexive over $\mathcal{M} \times \mathcal{M}$, where $\mathcal{M}$ is the set of all messages that may be generic broadcast, thus $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$ (PEDONE; SCHIPER, 2002). Hence, if $(m_1, m_2) \in \mathcal{C}$, then message m_1 conflicts with message m_2, and if $(m_1, m_2) \notin \mathcal{C}$, then message m_1 does not conflict (it commutes) with the message m_2 (PEDONE; SCHIPER, 2002). To simplify notation, throughout this work we write $m_1 \sim m_2$ to indicate that $(m_1, m_2) \in \mathcal{C}$ and $m_1 \nsim m_2$ otherwise. These primitives provide the following properties (PEDONE; SCHIPER, 1999; PEDONE; SCHIPER, 2002):

- **Validity**: if a correct process in Π $\boxed{\text{gb-Send } m \text{ to } \mathcal{D}}$, then all correct processes in $\mathcal{D}$ eventually $\boxed{\text{gb-Delivered } m}$;;
- **Agreement**: if a process in $\mathcal{D}$ $\boxed{\text{gb-Delivered } m}$, then all correct processes in $\mathcal{D}$ eventually $\boxed{\text{gb-Delivered } m}$
- **Integrity**: for any message m, every process in $\mathcal{D}$ $\boxed{\text{gb-Delivered } m}$ at most once, and only if m was previously $\boxed{\text{gb-Send } m \text{ to } \mathcal{D}}$ by a process in Π;
- **Partial Order**: if processes p_1, p_2 in $\mathcal{D}$ both $\boxed{\text{gb-Delivered } m_1}$ and $\boxed{\text{gb-Delivered } m_2}$, and $m_1 \sim m_2$, then p_1 and p_2 $\boxed{\text{gb-Delivered } m_1}$ and $\boxed{\text{gb-Delivered } m_2}$ in the same order.

The generic broadcast problem is generalization of atomic and reliable broadcast: when $\mathcal{C} = \mathcal{M} \times \mathcal{M}$, that is, all messages conflict, the problem reduces atomic broadcast; when $\mathcal{C} = \emptyset$, that is, no messages conflict, it reduces to reliable broadcast. Another problem, the Generalized Consensus (LAMPORT, 2005), goes even further and allows, for example, generalizing lease allocation (REZENDE, 2017). Here, however, we are more interested in a different generalization, allowing multicast to benefit from partial ordering.

2.3.2.5 Generic Multicast, Or The Goal

In this work, we focus on the generic multicast problem. A primitive for generic multicast combines the partial ordering of generic broadcast with the destination flexibility of multicast.

We define the generic multicast problem in terms of primitives $\boxed{\text{gm-Send } m \text{ to } \mathcal{G}}$, through which a process in Π can multicast a message m to every process in $\bigcup_{\mathcal{G}}$, $\mathcal{G} \subseteq \Gamma$; $\boxed{\text{gm-Delivered } m}$, in which a process $\bigcup_{\mathcal{G}}$ delivers a message m; and the conflict relation

$\mathcal{C}$, symmetric, non-reflexive over $\mathcal{M} \times \mathcal{M}$, thus $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$. An algorithm that solves the generic multicast must fulfill the following properties (ANTUNES, 2019; COELHO; SCHIPER; PEDONE, 2017):

- **Validity**: if a correct process in Π $\boxed{\text{gm-Send } m \text{ to } \mathcal{G}}$, $\mathcal{G} \subseteq \Gamma$, then $\forall p \in \bigcup_{\mathcal{G}}$ such that p is correct, eventually p $\boxed{\text{gm-Delivered } m}$;
- **Agreement**: if $\exists p \in \bigcup_{\mathcal{G}}$, $\mathcal{G} \subseteq \Gamma$, p $\boxed{\text{gm-Delivered } m}$, then every correct process in $\bigcup_{\mathcal{G}}$ eventually $\boxed{\text{gm-Delivered } m}$;
- **Integrity**: $\forall m \in \mathcal{M}$, $\forall p \bigcup_{\mathcal{G}}$, $\mathcal{G} \subseteq \Gamma$ p $\boxed{\text{gm-Delivered } m}$ at most once, and only if m was previously $\boxed{\text{gm-Send } m \text{ to } \mathcal{G}}$ by some process in Π;
- **Partial Order**: if processes p_1, $p_2 \in \Pi$ both $\boxed{\text{gm-Delivered } m_1}$ and $\boxed{\text{gm-Delivered } m_2}$, and $m_1 \sim m_2$, then p_1 $\boxed{\text{gm-Delivered } m_1}$ before $\boxed{\text{gm-Delivered } m_2}$, if, and only if, p_2 $\boxed{\text{gm-Delivered } m_1}$ before $\boxed{\text{gm-Delivered } m_2}$;
- **Acyclic Order**: the relation $\lesssim$ is acyclic, where for m_1, $m_2 \in \mathcal{M}$ and $m_1 \sim m_2$ then $m_1 \lesssim m_2$, if, and only if, there exists a process that $\boxed{\text{gm-Delivered } m_1}$ before $\boxed{\text{gm-Delivered } m_2}$.

As atomic multicast can solve atomic broadcast, generic multicast can solve generic broadcast, too. Also, generic multicast is a generalization of atomic and reliable multicast, only varying the conflict relation to achieve the desired behavior.

2.4 Temporal Logic of Actions

This section describes the *temporal logic of actions*, known as TLA. TLA provides mathematical foundations to specify and reason about concurrent systems (LAMPORT, 1994b; LAMPORT, 2002). Verifying the algorithm's correctness by writing the algorithm with a pseudo-code or a programming language is a task more difficult than reasoning about a one-page abstract algorithm written in mathematical notation (LAMPORT, 1994b). Programming languages have a difficult job to execute and can have details that are not explicit, while it could be easier with simple mathematical concepts (LAMPORT, 1994b).

Writing a system's formal specification takes effort, but some benefits include understanding the system better and having greater confidence in its operation (LAMPORT, 2002). There exists a gap between writing a specification and implementing an algorithm, where filling this gap by *supposing* how the system should behave can lead to implementing something other than the correct algorithm (CHANDRA; GRIESEMER; REDSTONE, 2007). The specification is not the final step; it is a tool to apply when appropriate. For example, during system design, use it to verify the interaction between the system's components (LAMPORT, 2002). With the ability to write a formal specification, developers have a new canvas to test ideas (NEWCOMBE et al., 2015).

The *correctness* of the system means that its properties are satisfied (LAMPORT, 1994b). We can represent a system with abstract objects to verify its correctness and use a model checker for invariant properties (YU; MANOLIOS; LAMPORT, 1999). To completely specify a system is also an activity of abstraction (LAMPORT, 1994a). For example, create an abstraction to separate the network layer from an algorithm specification. Learning how to abstract accurately, leaving only the essence of the algorithm, is a skill gained only through experience (LAMPORT, 2002).

This section describes the tools we use to formalize our algorithms. The algorithm's correctness does not depend on the formalism used to prove its correctness; it should be correct regardless (LAMPORT, 1994b). Remember that: *prose* is not a formal way to specify a system (LAMPORT, 2022), wherein the tool for such a task is formal methods, and we opt to use TLA.

2.4.1 Let There be Time

The system specification is a set of possible behaviors; a single *behavior* is a sequence of states; a *state* is an assignment of values to variables (LAMPORT, 2002). A single temporal formula F is an assertion of a system's behavior, evaluated as true or false; it is composed of elementary formulas using boolean operators and the unary $\Box$ operator (LAMPORT, 1994b; LAMPORT, 2002).

The boolean value a formula F assigns to behavior σ is denoted as $\sigma[[F]]$ (LAMPORT, 1994b). We say that σ satisfies F, if, and only if, $\sigma[[F]]$ equals true (LAMPORT, 2002). We can express the universe evolution as $\sigma_0 \rightarrow \sigma_1 \rightarrow ...$, where σ_n represents the state at instant n during behavior σ (LAMPORT, 1994b; LAMPORT, 2002). Different operators exist to validate the system during execution.

Machinery operators

To assert if any arbitrary temporal formula F is *always* valid. We start defining $\sigma[[\Box F]]$, to be true if, and only if, $\sigma_n \rightarrow \sigma_{n+1}$. Defining $\sigma^{+n} \triangleq \sigma_n \rightarrow \sigma_{n+1} \rightarrow ...$, as the suffix of σ removing the first n states, so $\sigma[[\Box F]]$ is true if, and only if, $\sigma^{+n}[[F]]$ is true for all n. Thus σ satisfies $\Box F$ if, and only if, every suffix σ^{+n} of σ satisfies F (LAMPORT, 2002).

$$\begin{aligned} \sigma^{+n} &\triangleq \sigma_n \rightarrow \sigma_{n+1} \rightarrow \sigma_{n+2} \rightarrow \dots \\ \sigma[[\Box F]] &\triangleq \forall\, n \in \texttt{Nat} : \sigma^{+n}[[F]] \end{aligned} \tag{1}$$

Equation (1) defines the temporal operator $\Box$. The formula $\Box F$ asserts that F is true at all times, reading as *always*, *henceforth*, or *from then on* (LAMPORT, 2002). There are other temporal formula classes, each class described in terms of boolean operators and the temporal operator $\Box$ (LAMPORT, 1994b). Another temporal operator is the $\Diamond$, read as *eventually*, and the formula $\Diamond F$, defined by $\neg\Box\neg F$ (LAMPORT, 1994b).

$$\sigma[[\Diamond F]] \triangleq \exists\, n \in \texttt{Nat} : \sigma^{+n}[[F]] \tag{2}$$

The $\Diamond$ operator asserts that F is not always false or that F is true at some time (LAMPORT, 1994b; LAMPORT, 2002). Equation (2) specifies the $\Diamond$ operator. A behavior satisfies $\Diamond F$ if, and only if, F is valid at some time during the behavior (LAMPORT, 1994b).

Combining $\Diamond$ and $\Box$, we have two new operators. The first is the $\Box\Diamond$, which read as *infinitely often* (LAMPORT, 1994b). The formula $\Box\Diamond F$ asserts that at all times, either F is valid then or at some time later (LAMPORT, 2002), formally written in Equation (3) (LAMPORT, 1994b). The other operator is the $\Diamond\Box$, which reads and asserts that $\Diamond\Box\, F$ is *eventually always* valid. A behavior satisfies $\Diamond\Box F$ if, and only if, after some time, it is always true from that time on (LAMPORT, 1994b). Formally written in Equation (4).

$$\sigma^{+(n+m)} \triangleq \sigma_{n+m} \rightarrow \sigma_{n+m+1} \rightarrow \sigma_{n+m+2} \rightarrow \ldots$$

$$\sigma[[\Box\Diamond F]] \triangleq \forall\, n \in \texttt{Nat} : \exists\, m \in \texttt{Nat} : \sigma^{+(n+m)}[[F]] \tag{3}$$

$$\sigma[[\Diamond\Box F]] \triangleq \exists\, n \in \texttt{Nat} : \forall\, m \in \texttt{Nat} : \sigma^{+(n+m)}[[F]] \tag{4}$$

The last temporal operator is $\leadsto$. For any two temporal formulas, F and G, it is written as $F \leadsto G$, or, in other words, $\Box(F \implies \Diamond G)$, asserting that any time that F is true, then eventually, G is also true (LAMPORT, 1994b). This operator is also transitive, meaning that if $F \leadsto G$ and $G \leadsto H$ are both satisfied, then $F \leadsto H$ is also satisfied (LAMPORT, 1994b). More formally, for any temporal formulas F and G (LAMPORT, 2002):

$$\sigma[[F \leadsto G]] \triangleq \forall\, n \in \texttt{Nat} : (\sigma^{+n}[[F]]) \implies (\exists\, m \in \texttt{Nat} : \sigma^{+(n+m)}[[G]])$$

We have a complete framework to assert a system's behavior during execution. We can represent time passing when specifying our algorithms, but some systems may perceive the passage of time differently. For example, a specification for a clock that displays hours, minutes, and seconds implements one that shows hours and minutes only, but the former sees time differently from the latter. For this specification to be valid, the systems must be able to do nothing; if the minute changes in every step, then no clock displaying seconds exists (LAMPORT, 2002).

Falters' act

The TLA specification represents the complete universe, whereas, in this universe, the system exists with all other systems. For example, in mathematical terms, the formula $f(x) = x^2 + x + 1$ does not represent the universe strictly for x; it is the whole universe, but with a focus on the x variable (LAMPORT, 1994a). Since there is a complete universe on the specification, some parts can evolve while others remain unchanged.

In TLA, a *stuttering step* describes a step in which the system remains the same (LAMPORT, 1994a). That is, the system must be capable of not changing while the universe is still going. An *action* represents the relation between old and new variable values (LAMPORT, 1994b). A *state function* is an ordinary nonboolean expression that can contain variables and constants (LAMPORT, 1994b; LAMPORT, 2002). For any action $\mathcal{A}$, every state function f, to denote that a system complies with changing and not changing as well, is written as (LAMPORT, 1994a):

$$[\mathcal{A}]_f \ \triangleq \ \mathcal{A} \vee (f' = f)$$

A step satisfies $[\mathcal{A}]_f$ (read as *square* $\mathcal{A}$ *sub* f) if, and only if, the action $\mathcal{A}$ is valid or the state f does not change (LAMPORT, 1994a), where, in such cases, it stuttered, the universe changed while the specified system did not. To assert that in every step, $\mathcal{A}$ is either satisfied or f is unchanged, represented by the formula $\Box[\mathcal{A}]_f$ (LAMPORT, 1994a). Through these steps, a system can, every time, execute only $f' = f$, meaning the system never changes, never making any progress. Fairness can ensure progress in the specification (LAMPORT, 1994a).

The fairness

A specification can use strong (SF) and weak (WF) fairness (LAMPORT, 1994b; LAMPORT, 1994a). Informally, WF asserts that action $\mathcal{A}$ is either eventually executed or impossible, even if impossible only briefly. SF asserts that the action is either eventually executed or eventually becomes always impossible. Writing both of these informal descriptions as (LAMPORT, 1994b):

$$\begin{aligned} \text{WF} &: (\Diamond\, \text{executed}) \vee (\Diamond\, \text{impossible}) \\ \text{SF} &: (\Diamond\, \text{executed}) \vee (\Diamond\Box\, \text{impossible}) \end{aligned}$$

The "executed' means that action $\mathcal{A}$ is *enabled*, where it is enabled iff there is a state t satisfying $\mathcal{A}$, expressed as $\Diamond\langle\mathcal{A}\rangle_f$. Dissecting the expression, $\langle\mathcal{A}\rangle_f$ is an $\mathcal{A}$ step that changes the values in f (LAMPORT, 1994b); with the $\Diamond$ operator, we have that: eventually, every step changes the state. The "impossible" means we can not take step

$\mathcal{A}$ with state f. That is, action $\mathcal{A}$ is not enabled, written as $\neg \mathit{Enabled}\langle \mathcal{A} \rangle_f$ (LAMPORT, 1994b). Therefore, expressed by the formulas (LAMPORT, 1994b):

$$\begin{aligned} \mathrm{WF}_f(\mathcal{A}) &\triangleq (\Box\Diamond\langle \mathcal{A} \rangle_f) \vee (\Box\Diamond\neg\, \mathit{Enabled}\langle \mathcal{A} \rangle_f) \\ \mathrm{SF}_f(\mathcal{A}) &\triangleq (\Box\Diamond\langle \mathcal{A} \rangle_f) \vee (\Diamond\Box\neg\, \mathit{Enabled}\langle \mathcal{A} \rangle_f) \end{aligned}$$

Since $\Diamond\Box F \implies \Box\Diamond F$, thus $\mathrm{SF}_f(\mathcal{A}) \implies \mathrm{WF}_f(\mathcal{A})$ (LAMPORT, 1994b). Whenever written $\mathrm{SF}_f(\mathcal{A})$ or $\mathrm{WF}_f(\mathcal{A})$ implies that $f' \neq f$, at any action that $\mathcal{A}$ is enabled, then the state f changed (LAMPORT, 1994a). All in all, for any step, it either stutters or changes.

Liveness and safety

Programs can show undesirable behavior. The specification is a description of what the system is supposed to do (LAMPORT, 2002), whereas, for the algorithm to be correct, it must satisfy the desired properties (LAMPORT, 1994b). The system's *safety* properties assert that bad things never happen (ALPERN; SCHNEIDER, 1987), meaning the system never enters an unacceptable state (OWICKI; LAMPORT, 1982). Some safety examples are that a program never enters a situation where progress is impossible; two different processes can not access a critical section simultaneously (OWICKI; LAMPORT, 1982). Safety properties do not require fairness (OWICKI; LAMPORT, 1982).

Safety by itself does not require the system to do something (LAMPORT, 2002), meaning that, by doing nothing, we do not do anything wrong. Employing *liveness* properties, we can assert that something good eventually does happen (OWICKI; LAMPORT, 1982; ALPERN; SCHNEIDER, 1987). Liveness properties that should eventually occur are, for example, answering each request or a message reaching the destination (OWICKI; LAMPORT, 1982). Many systems only guarantee liveness with fairness (OWICKI; LAMPORT, 1982).

$$\Phi_1 \triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{vars} \tag{5}$$

$$\Phi_2 \triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{vars} \wedge \mathit{Liveness} \tag{6}$$

A TLA specification has the format of the formula in Equation (5), which is a safety property (LAMPORT, 1994b). It asserts that system starts satisfying *Init* and only takes steps $[\mathit{Next}]_{vars}$ (LAMPORT, 1994a). Equation (6) strengthens Equation (5) adding liveness property. *Liveness* is a conjunction of formulas using fairness, with action $\mathcal{A}$, $\mathrm{WF}_{vars}(\mathcal{A})$ and $\mathrm{SF}_{vars}(\mathcal{A})$ (LAMPORT, 2002). Decomposing Equation (6), *Init* constrains the system's initial state, $[\mathit{Next}]_{vars}$ constrains the steps it may take, and *Liveness* what must eventually happen (LAMPORT, 2002)

All properties are equal, but some properties are more equal than others. Liveness property is philosophically important, but, in practice, safety property is paramount

(LAMPORT, 2002). The goal when writing a specification is to avoid errors, so in comparison with liveness, safety properties bring more benefits to the table (LAMPORT, 2002). But, since the liveness properties are easy enough to write and constitute a small part of the specification, we might as well write them down (LAMPORT, 2002).

What a wonderful system

We can express the system's properties using temporal logic. A program Ψ has property F, expressable as $\Psi \implies F$, asserting that every behavior that satisfies Ψ will satisfy property F (LAMPORT, 1994b). We use these properties to explain two popular classes of properties, *invariance* and *eventuality* (LAMPORT, 1994b). The properties' proofs use axioms and proof rules. A proof rule $\frac{F,G}{H}$ asserts that $\vdash F$ and $\vdash G$ imply $\vdash H$ (LAMPORT, 1994b).

The formula $\Box P$, where P is a predicate, expresses an invariance property (LAMPORT, 1994b). For example, P can assert that at most one process is in the critical section simultaneously; or that the program never enters a state in which progress is impossible. Rule INV1 in Equation (7b) proves that a program satisfies an invariance property $\Box P$ (LAMPORT, 1994b).

$$\text{LATTICE.}\ \succ \text{ a well-founded partial order on a set } S$$
$$\frac{F \wedge (c \in S) \implies (H_c \leadsto (G \wedge \exists d \in S : (c \succ d) \wedge H_d)}{F \implies ((\exists c \in S : H_c) \leadsto G)} \tag{7a}$$

$$\text{INV1.}\ \frac{I \wedge [\mathcal{N}]_f \implies I'}{I \wedge \Box[\mathcal{N}]_f \implies \Box I} \tag{7b}$$

Eventuality properties assert that something eventually happens (LAMPORT, 1994b). For example, a program terminates at some point (LAMPORT, 1994b). There are different ways to express these properties, which are reducible to formulas of form $P \leadsto Q$. The reduction is proven using the rule LATTICE and temporal reasoning (LAMPORT, 1994b).

The invariance and eventuality are essential to check the system's properties. Using these properties to verify if an algorithm holds all the guarantees; if a system is designed correctly and fulfilling all requirements.

2.4.2 A Useful Model

We can specify systems in TLA using the TLA$^+$ language. TLA$^+$ is a language where TLA meets first-order logic and Zermelo-Fraenkel set theory (YU; MANOLIOS;

LAMPORT, 1999). TLA$^+$ can describe high-level correctness properties to the low-level design of a system (YU; MANOLIOS; LAMPORT, 1999). It is available with some tools, which include a model checker (LAMPORT, 2002; LAMPORT, 2021b). The model checker, known as TLC, is used for finding errors in TLA$^+$ specifications (LAMPORT, 2002).

The TLC model checker handles a subclass of TLA$^+$ specifications, where it might not operate a large model of a specification, but it should deal with most real-world system specifications (YU; MANOLIOS; LAMPORT, 1999). TLC's input is a TLA$^+$ module, assuming a formula in form $\Phi \triangleq Init \wedge \Box[Next]_{vars} \wedge Liveness$, the same as Equation (6) on page 35, and a configuration file describing the specification formula and properties to check (LAMPORT, 2002). The most effective way to find errors is by verifying the system's invariant properties (YU; MANOLIOS; LAMPORT, 1999; LAMPORT, 2002).

Internally, TLC maintains an explicit state representation, not using a symbolic approach (YU; MANOLIOS; LAMPORT, 1999). TLC has two data structures: a set *seen* of known reachable states and a FIFO queue containing elements of *seen* with the successor states not checked (YU; MANOLIOS; LAMPORT, 1999). The values in *seen* are the state's 64-bit fingerprint, and in the queue are the actual states (YU; MANOLIOS; LAMPORT, 1999).

When verifying a model, TLC generates and checks all possible states that satisfy the *Init* predicate, populating the queue and *seen* with these states (YU; MANOLIOS; LAMPORT, 1999). Then, TLC rewrites the next-state relation *Next* as a disjunction of every smallest subaction possible (YU; MANOLIOS; LAMPORT, 1999). A set of workers is then launched and repeatedly do:

- Remove a state s from the front of the queue;
- For each subaction $\mathcal{A}$, generate every next state t where the pair s and t satisfy $\mathcal{A}$.

TLC reports a deadlock when no next state t exists and reports an error if t does not satisfy an invariant or when s does not have the next state (YU; MANOLIOS; LAMPORT, 1999). For every next state t, the workers do (YU; MANOLIOS; LAMPORT, 1999):

- If t is not in *seen*, check if t satisfies the invariant;
- If t in *seen*, add t to *seen* pointing to s;
- If t satisfies the constraint, add t at the end of the queue.

TLC evaluates expressions to check the specification (LAMPORT, 2002). TLC evaluates the expressions left-to-right, similar to how a person would mentally evaluate (LAMPORT, 2002). We must pay attention to the evaluation process. For example, in the logically equivalent formulas, $(x \neq \langle\rangle) \land (x[1] = 0)$ and $(x[1] = 0) \land (x \neq \langle\rangle)$, TLC evaluates the former correctly, whereas the latter raises an error (LAMPORT, 2002). TLC's evaluation process is (LAMPORT, 2002):

- For a formula $p \land q$, evaluates p and, if it equals `TRUE`, then evaluates q;
- For a formula $p \lor q$, evaluates p and, if it equals `FALSE`, then evaluates q;
- For a formula $p \implies q$, evaluates as $\neg p \lor q$;
- For `IF` p `THEN` e_1 `ELSE` e_2, evaluates p, then evaluates either e_1 or e_2.

For a set S, TLC enumerates all elements of S in some order and evaluates the expression substituting with one value at a time (LAMPORT, 2002). When handling sets, TLC declares an error if it is not obviously finite. TLC similarly evaluates the following expressions:

$$\begin{array}{ccc} \exists x \in S : p & \forall x \in S : p & \texttt{CHOOSE}\ x \in S : p \\ \{x \in S : p\} & \{e : x \in S\} & [x \in S \mapsto e] \\ \texttt{SUBSET}\ S & \texttt{UNION}\ S & \end{array}$$

TLC can evaluate a temporal formula F if, and only if, F is *nice* and can evaluate the formulas that compose F (LAMPORT, 2002). The temporal formula F is nice if, and only if, it is a conjunction of formulas belonging to the classes of *state predicates*, an ordinary boolean-valued expression, with no prime nor $\Box$ operator; *invariance formulas*, such as $\Box P$, where P is a state predicate; *box-action formulas*, such as $\Box[\mathcal{A}]_v$, where $\mathcal{A}$ is an action and v is a state function; and *simple temporal formulas* (LAMPORT, 2002). A simple temporal formula is composed of *temporal state formulas* and *simple action formulas* by applying *simple Boolean operators* (LAMPORT, 2002):

- **Simple Boolean operators**: consist of $\land$, $\lor$, $\neg$, $\implies$, $\equiv$, `TRUE` and `FALSE` with quantification over finite, constant sets;
- **Temporal state formula**: composed from state predicates by applying *simple Boolean operators* and temporal operators $\Box$, $\Diamond$, and $\leadsto$;
- **Simple action formula**: with the action $\mathcal{A}$ and state function v, is one of $\mathrm{WF}_v(\mathcal{A})$, $\mathrm{SF}_v(\mathcal{A})$, $\Box\Diamond\langle\mathcal{A}\rangle_v$, and $\Diamond\Box[\mathcal{A}]_v$.

LAMPORT gives some hints on effectively using TLC. Start with a reduced specification, find errors early, and then run TLC on larger models. A successful verification should raise suspicion; the finite model can hide liveness problems, as doing nothing can satisfy safety properties. Check properties that should find a violation, and verify as many invariance properties that make sense.

There exists much more for TLA, TLA$^+$, and TLC. The proof system, known as TLAPS, which we did not explore in this work. Advanced topics, such as composing specifications and specifications for real-time systems (LAMPORT, 2002).

2.5 Golang

This section discusses the Golang programming language, henceforth Go, used to implement Generic Multicast 1 algorithm. The Go language is a general-purpose, garbage-collected, compiled system programming language, which provides built-in features for concurrent programming (GOLANG, 2021b). Go follows a design for multi-threading applications, providing lightweight threads and explicit message passing (TU et al., 2019). The language is not overly complex and contains multiple features for implementing and testing the system. In this section, we discuss the concurrency features available.

Go's concurrency model originated from the Communicating Sequential Processes concurrency model, created by Tony Hoare (BUTCHER; FARINA, 2016; GOLANG, 2022a). Concurrency in Go is cheap, with two principal components that make this model work, the *goroutine* and *channels*, with a motto, "do not communicate by sharing memory; instead, share memory by communicating" (GOLANG, 2022a). The language encourages sharing values using channels instead of sharing memory between threads, believing that explicit message-passing is less error-prone (TU et al., 2019).

Goroutine is a cheap, lightweight user-level thread that executes concurrently along other goroutines in the same address space (GOLANG, 2022a). The Go runtime manages and maps the routines to OS threads in an M:N model, multiplexed to keep running, where one can wait for a resource and others continue working without blocking (BUTCHER; FARINA, 2016; GOLANG, 2022a; TU et al., 2019). Each routine costs a little more than stack space allocation and growing as needed (GOLANG, 2022a).

A channel is a concurrency primitive to send and receives data, passing values between routines (GOLANG, 2022a). Channels primitives, when used with good judgment, can help to write concise, correct programs (GOLANG, 2022a). Sharing by communicating is encouraged, but not enforced, being possible to synchronize goroutines in a conventional way using locks, conditions, and atomic operations (TU et al., 2019).

In summary, Go provides tools to ease the development of concurrent programs, making it a good fit for the current prototype implementation work. There is more that the language can offer, which is not detailed here. A quick list of features includes a low-latency garbage collector, compilation to native code, recently added support to generic types, and an ecosystem with multiple tools and libraries.

Chapter 3

Related Work

Our work has both theoretical and practical contributions, and in this chapter, we present and discuss works related to ours in these aspects. We start with Section 3.1, revisiting some multicast algorithms and the family lineage up to the algorithms we developed here. Section 3.2 discusses the formal verification of group communication and agreement algorithms. We conclude with Section 3.3, with a discussion of the implementations of these algorithms.

3.1 Multicast History

Skeen's algorithm is an Atomic Multicast algorithm for failure-free environments and has inspired many other works since. First referenced in (BIRMAN; JOSEPH, 1987) as an unpublished work, wherein the algorithms of our work descend from the same lineage.

In the protocol, each message has an assigned timestamp. The timestamp determines the delivery order between messages (SCHIPER; PEDONE, 2007) with an initial value as an assignment of the participating processes' local clocks. The initiator, that is, the process that first sends the message, acts as a coordinator in the procedure to decide the message's final timestamp. During the algorithm, participating processes maintain two sets, *Undeliverable* and *Deliverable*, to control the messages' state in the agreement of the timestamp value. The algorithm follows these five steps (FRITZKE et al., 1998; ANTUNES, 2019):

1. On the invocation of $\boxed{\text{am-Send } m \text{ to } \mathcal{G}}$, the initiator process $p \in \Pi$ will $\boxed{\text{Send } m \text{ to } q}$, $\forall q \in \mathcal{G}$;

2. Each process $q \in \mathcal{G}$ that $\boxed{\text{Received } m \text{ from } p}$, increase its clock, assigns a timestamp ts to be the current clock's value, adds $\langle m, ts\rangle$ to *Undeliverable*, and $\boxed{\text{Send } \langle m, ts\rangle \text{ to } p}$;

3. After the coordinator p $\boxed{\text{Received } \langle m, ts\rangle \text{ from } q}$, $\forall q \in \mathcal{G}$, it defines the maximum timestamp received as the definitive timestamp ts_f for m and $\boxed{\text{Send } \langle m, ts_f\rangle \text{ to } q}$,

$\forall q \in \mathcal{G}$;

4. For every $q \in \mathcal{G}$, on $\boxed{\text{Received } \langle m, ts_f \rangle \text{ from } p}$, remove m from *Undeliverable*, insert $\langle m, ts_f \rangle$ into *Deliverable*;

5. Each process $q \in \mathcal{G}$ will $\boxed{\text{am-Delivered } m}$, for every $\langle m, ts_f \rangle \in$ *Deliverable*, where the tuple $\langle m, ts_f \rangle$ is the smallest of all other tuples in *Deliverable* and *Undeliverable* and removing $\langle m, ts_f \rangle$ from *Deliverable*.

Figure 1 shows a successful execution of the protocol. We have processes p_1, p_2, p_3, and p_4 handling messages m_1 and m_2. Message m_1's destination is $\mathcal{G}_{m_1} = \{p_1, p_2, p_3\}$ and m_2's $\mathcal{G}_{m_2} = \{p_2, p_3, p_4\}$. This example has the multicast aspect in evidence, where processes p_2 and p_3 deliver messages in the same order, while p_1 and p_4 deliver when ready.

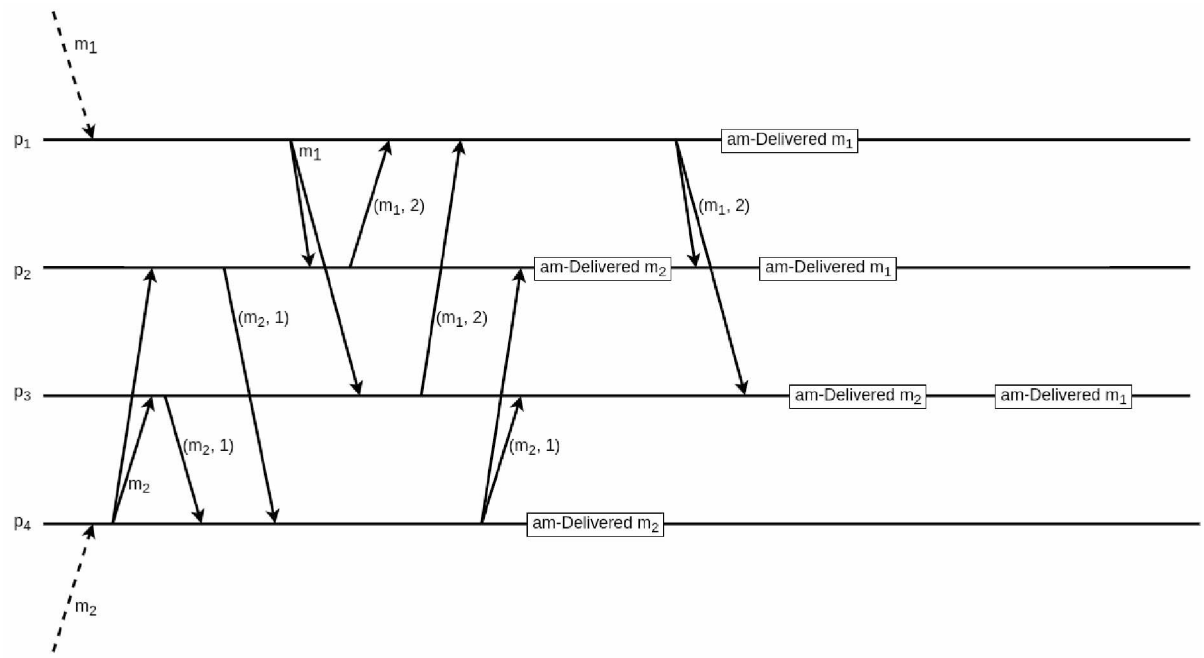

Figure 1 – Happy path execution.

In some executions, the timestamp is enough to order message delivery, but, in some cases, timestamps tie. Figure 2 depicts a timestamp tie. For $\mathcal{G} = \{p_1, p_2\}$ and messages m_1 and m_2, p_1 is the m_1 coordinator, and p_2 is m_2's. The proposals from p_1 and p_2 are $(\langle m_1, 1\rangle, \langle m_2, 2\rangle)$ and $(\langle m_2, 1\rangle, \langle m_1, 2\rangle)$, respectively. The decided timestamp is 2 for both messages because the coordinator selects the highest value. For processes to deliver messages in the same order, they must be able to sort messages deterministically to break timestamp ties.

In future works, FRITZKE et al. extended Skeen's algorithm to make it fault-tolerant. Fritzke's algorithm uses a replication approach, dealing with groups of processes instead

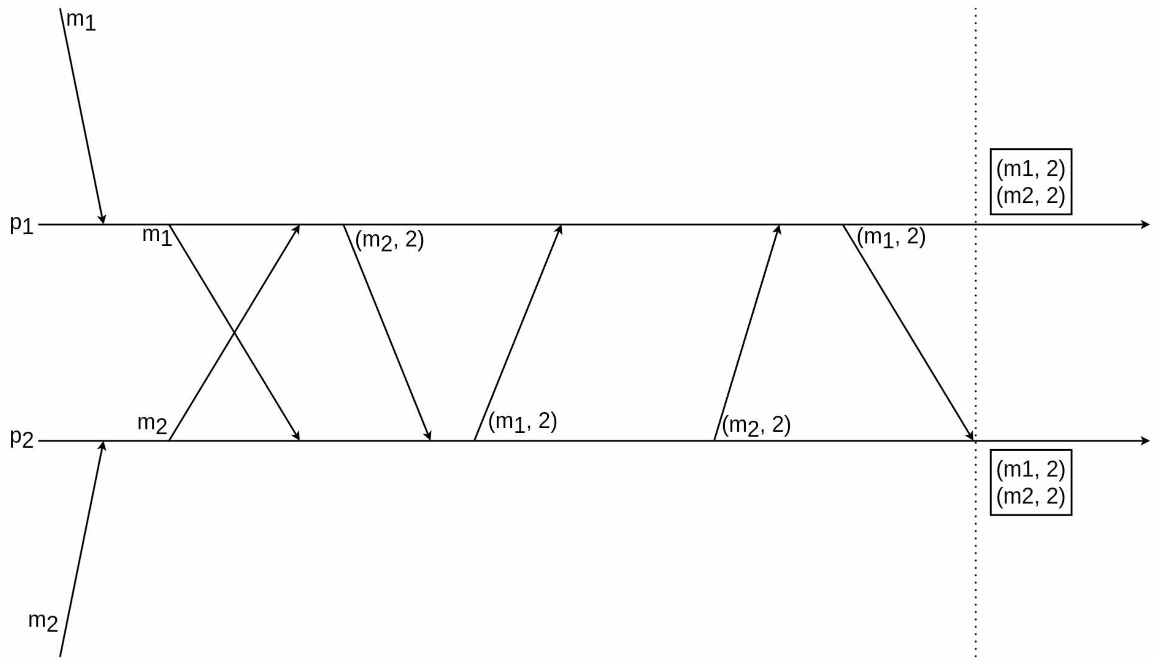

Figure 2 – Timestamp tie (ANTUNES, 2019).

of processes. The algorithm works in a different environment, where processes may crash. Every group has a majority of correct members and has a failure detector of class $\diamond\mathcal{S}$ attached (FRITZKE et al., 1998). When groups have a single process, the algorithm reduces to Skeen's algorithm (FRITZKE et al., 1998).

Fritzke's algorithm works like Skeen's, using timestamps to order message delivery. The algorithm requires a uniform Reliable Multicast primitive at the start. Since it handles groups, each timestamp proposal comes from a group. For each message, it is necessary two consensus rounds: the first to agree on the group's proposal and the second to the final timestamp. All consensus rounds are local to a single group, not involving processes in distinct groups, a property known as *locality* (FRITZKE et al., 1998).

SCHIPER; PEDONE further extend Fritzke's algorithm. The need for a uniform version of Reliable Multicast primitive is no more, while still guaranteeing properties as strong as Fritzke's version (SCHIPER; PEDONE, 2007). When $\boxed{\text{am-Send } m \text{ to } \mathcal{G}}$ and $|\mathcal{G}| = 1$, messages can receive the final timestamp and are ready for delivery, removing the need for a second consensus round.

In a later work, AHMED-NACER; SUTRA; CONAN studied the convoy effect in Atomic Multicast primitives. The convoy effect is a phenomenon in which the delivery of one or more messages delays other ones (BLASGEN et al., 1979; AHMED-NACER; SUTRA; CONAN, 2016). To circumvent the performance degradation that the convoy effect causes, the authors propose to use the messages' semantics (AHMED-NACER; SUTRA; CONAN, 2016). One of the results is a Generic Multicast algorithm built on top of Skeen's algorithm.

The work of ANTUNES applies AHMED-NACER; SUTRA; CONAN's proposal of using the messages' semantics to the Atomic Multicast algorithms, resulting in three Generic Multicast algorithms, *Generic Multicast 0*, *Generic Multicast 1*, and *Generic Multicast 2*. All ANTUNES' algorithms have a generalized approach that creates a partial order for message delivery. Generic Multicast 0 extends Skeen's algorithm from AHMED-NACER; SUTRA; CONAN's work, working in a failure-free environment (ANTUNES, 2019). Generic Multicast 1 builds upon SCHIPER; PEDONE's extension of FRITZKE et al.'s algorithm, using the same environment where processes may crash. Lastly, Generic Multicast 2 uses the lessons from Generic Multicast 0 and Generic Multicast 1, resulting in an algorithm where all group communication primitives are generalized (ANTUNES, 2019).

Our work continues ANTUNES' work. Generic Multicast 0, Generic Multicast 1, and Generic Multicast 2 all lack formal verification. For the current work, we write TLA$^+$ specifications for all of ANTUNES' algorithms, and we propose our improvements. The original algorithms had subtle problems that went unnoticed without validation. We will discuss our findings and solutions in the next chapter. Figure 3 displays the multicast algorithms' family tree.

The work of PEDONE; SCHIPER is also a cornerstone and inspiration for our work. We use a preliminary version of (PEDONE; SCHIPER, 2002). PEDONE; SCHIPER presents the notion of strictness and delivery latency, which we apply to our work, too.

3.2 Algorithms in Theory

Academia and industry have been using TLA$^+$ in a plethora of projects (LAMPORT, 2021a). We will start with TLA$^+$ use in academia and then its use in industry. REZENDE has a work focused on the Generalized Consensus problem (LAMPORT, 2005). REZENDE specifies the Generalized Paxos in TLA$^+$ and implements an instance that solves a variation of the distributed lease coordination.

The work of CAMARGOS contributes new consensus algorithms and an abstraction called *Log Service*, among other contributions. The Log Service abstracts the atomicity and durability problems in transaction termination (CAMARGOS, 2008). Both the algorithm and Log Service have a TLA$^+$ specification.

ONGARO's work introduces Raft, an algorithm to solve the Atomic Broadcast problem. Multiple production systems rely on the Raft algorithm; correctness is a crucial requirement for such an algorithm. ONGARO wrote a TLA$^+$ specification and proof for the algorithm. The manual proof relies on the TLA$^+$ specification, where there exist lemmas that follow directly from it. ONGARO's work is also interesting because it references the hardships of verifying larger models in TLA$^+$. Model checking larger models is a difficult task in means of time and storage necessary.

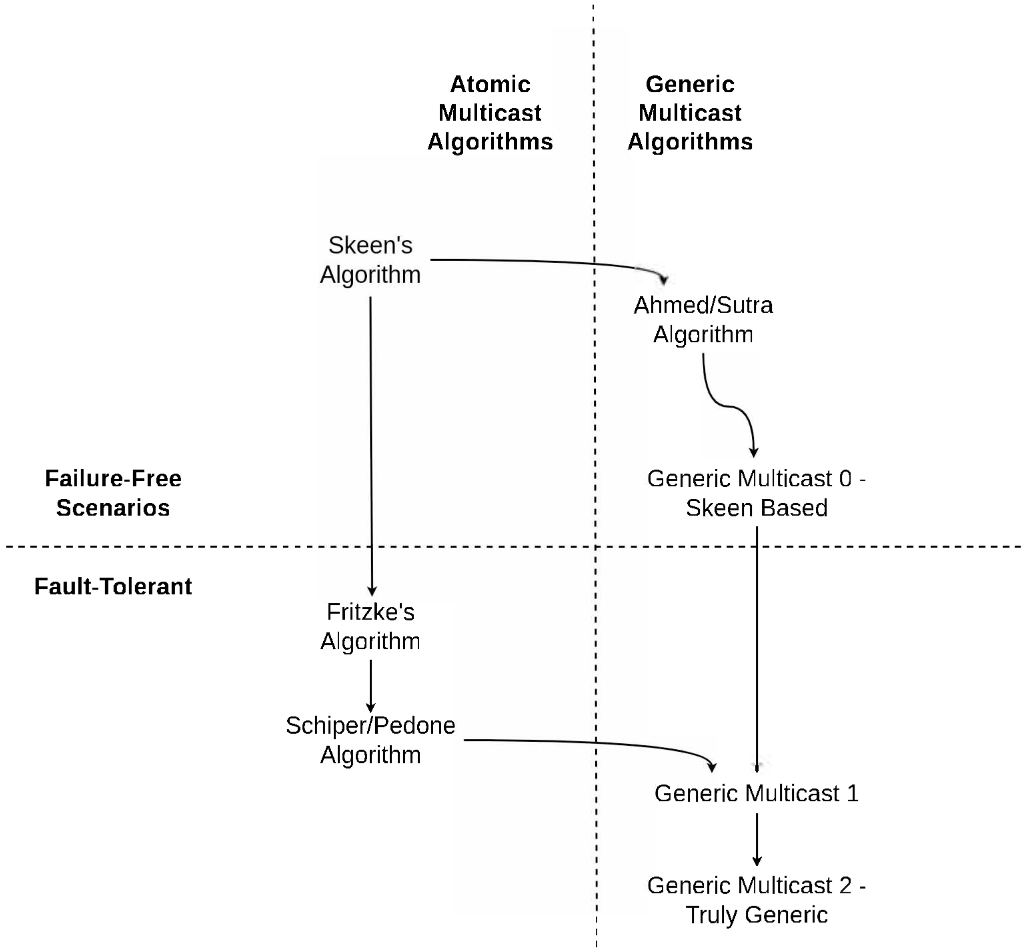

Figure 3 – Extensions of multicast algorithms (ANTUNES, 2019).

The industrial use of TLA$^+$ varies from verifying algorithms to verifying running system designs. In AWS, engineers use TLA$^+$ to specify algorithms and design of large distributed systems. NEWCOMBE et al. published a report describing the AWS use and adoption process. The authors report that TLA$^+$ helps avoid problems reaching production, finding subtle bugs in algorithms, bugs that escape reviews and would be difficult to find otherwise. The authors find that thinking in safety and liveness terms is less error-prone than the usual development approach of imagining what could go wrong and starting to patch possible scenarios. Writing a TLA$^+$ specification gives more confidence in the system's design correctness, giving space to engineers to propose improvements and check *what-if* scenarios. Applying such methods pays in the system's lifetime, providing a faster time-to-market for products without giving-up quality and correctness.

TLA$^+$'s first use was to verify hardware model (LAMPORT, 2021a). BEERS' work describes how Intel applied formal verification in early cycles before the target RTL to verify a coherence protocol and its implementation. In their experience, the author concludes that the early iterations kept problems out of the RTL, giving engineers a solid microarchitecture and making verification after the RTL phase more efficient. In unpublished papers but open-source contributions, Open Networking Foundation and Atomix applied TLA$^+$ to multiple and varying types of problems[1], naming a few, a distributed lock, adding custom functionalities to Raft, verifying systems design, and experiments to verify the implementations adhere to a specification. Microsoft applied TLA$^+$ to write specifications for the different consistency operations provided by their distributed databases[2].

Our work uses TLA$^+$ to verify problems in ANTUNES' algorithms. Similar to the examples in this section, we apply TLA$^+$, meaning that we are not proposing something novel to TLA$^+$. In this aspect, our goal is only to verify problems and apply fixes.

3.3 Algorithms in Practice

Leaning toward implementation, we have some work focusing on applying formal methods to the development process. A proposition from SYSTEMS is called *Verification-Driven Development*, primarily focusing on distributed and concurrent systems. Researchers and engineers collaborate in a refinement cycle with multiple steps to solve a programming problem.

The development starts with a high-level description of the problem, not involving proofs, only prose to introduce the problem. With the high-level description of the problem, it's time for the system definitions, failures, communication, processes, synchrony, and safety and liveness properties. Having all these definitions is possible to formulate an algorithm. The focus is on developing a complete TLA$^+$ specification with all the properties and invariants. Using the algorithm TLA$^+$ specification, engineers can write an implementation specification. The implementation specification details the process behavior and includes how the algorithm specification reduces to the implementation specification. The last step is coding!

There exists a gap between specification and implementing an algorithm, and even though the specification is correct, the translation to a programming language can have bugs. In the work of BORNHOLT et al., the authors report the use of "lightweight formal methods" to validate a storage implementation, meaning the use of the appropriate tool for each problem, easy to apply by engineers, and the possibility to evolve the models and specifications. The authors took this approach because they needed more flexibility to verify different properties, like API calls and crash consistency, and were willing to give up some guarantees that a formal specification offer.

The approach has three main elements. An executable for a basic model conforming to the specification, the model is used as a reference. Use the reference model to check the actual implementation, applying tools that best fit the case for functional correctness, concurrency, and crash consistency. The reference model evolves as the project evolves,

[1] ONF/Atomix usage.
[2] Microsoft CosmosDB presentation.

educating engineers to use and extend the methods during development. Unfortunately, such an approach does not guarantee that problems do not exist, but it has effectively avoided problems of reaching production, and engineers can integrate the formal methods during development.

Our work does not focus on how to apply formal methods during implementation. We do not make any proposal of this kind whatsoever. We follow an approach similar to the verification-driven development to implement a prototype for the Generic Multicast 1 algorithm. We implemented the algorithm and the specification simultaneously, one helping the other.

Chapter 4

Correctness Development

This chapter contains our contributions to the formalization and verification of the algorithms proposed by ANTUNES. These include identifying problems through model checking, corrections, and the experimental validation of the corrected algorithms. To simplify the presentation, we only display excerpts of the specifications in this chapter, where the complete TLA$^+$ specifications are available in Appendix A.

Common to all Generic Multicast algorithms discussed is that all messages sent through the algorithm are associated with a timestamp and that the algorithm uses the timestamp to deliver messages in a partial order. Timestamps are defined based on a conflict relation (PEDONE; SCHIPER, 1999): conflicting messages either have different timestamps and are delivered in timestamp order, or the timestamps are equal and are delivered based on some deterministic ordering with respect to each other (ANTUNES, 2019). All messages in the algorithm belong to a set $\mathcal{M}$, which has a strict total order.

This chapter includes a description of all algorithms and how they work. Each algorithm is a step towards a complete generalized form. We conclude the chapter with our experience using TLA$^+$, create a link between the TLA concepts presented in Section 2.4, and include additional properties these algorithms provide. We only define propositions for these properties, leaving the proofs for future work.

4.1 Generic Multicast 0

The first algorithm verified with TLA$^+$ was Generic Multicast 0 (ANTUNES, 2019), based on Skeen's algorithm for failure-free systems (BIRMAN; JOSEPH, 1987). Since the algorithm works on failure-free systems, it serves as a gentle first contact with Generic Multicast (ANTUNES, 2019).

The algorithm associates the multicast messages with tentative timestamps derived from logical clocks. The algorithm uses a conflict relation when assigning a timestamp, increasing the processes' clock only when necessary, trying to keep the timestamp value as low as possible (ANTUNES, 2019). The multicast initiator coordinates the process to determine a final timestamp; we also refer to the initiator as the message coordinator. Algorithm 4.1 presents the pseudo-code, where all procedures have an atomic execution.

Algorithm 4.1 Generic Multicast 0

Variables:
$K \leftarrow 0$
$\mathit{Pending} \leftarrow \emptyset$
$\mathit{Delivering} \leftarrow \emptyset$
$\mathit{Delivered} \leftarrow \emptyset$
$\mathit{PreviousMsgs} \leftarrow \emptyset$

procedure GM-SEND$(m, \mathcal{G})$ ▷ Process p
    let $m.d = \mathcal{G}$
    for all $q \in m.d$ **do**
        Send $\langle S0, m \rangle$ to q

procedure ASSIGNTIMESTAMP ▷ Process q
when: Received $\langle S0, m \rangle$ from p
    if $\exists\, m_i \in \mathit{PreviousMsgs} : m \sim m_i$ **then**
        $K \leftarrow K + 1$
        $\mathit{PreviousMsgs} \leftarrow \emptyset$
    $\mathit{PreviousMsgs} \leftarrow \mathit{PreviousMsgs} \cup \{m\}$
    $\mathit{Pending} \leftarrow \mathit{Pending} \cup \{\langle m, K \rangle\}$
    Send $\langle S1, m, K \rangle$ to p

procedure COMPUTESEQNUMBER ▷ Process p
when: $\forall\, q \in m.d$: Received $\langle S1, m, ts \rangle$ from q
    $ts_f \leftarrow \max(\{ts : \text{Received } \langle S1, m, ts \rangle\})$
    for all $q \in m.d$ **do**
        Send $\langle S2, m, ts_f \rangle$ to q

procedure ASSIGNSEQNUMBER ▷ Process q
when: Received $\langle S2, m, ts_f \rangle$ from p $\wedge \langle m, \_ \rangle \in \mathit{Pending}$
    if $ts_f > K$ **then**
        if $\exists\, m_i \in \mathit{PreviousMsgs} : m \sim m_i$ **then**
            $K \leftarrow ts_f + 1$
            $\mathit{PreviousMsgs} \leftarrow \emptyset$
        else
            $K \leftarrow ts_f$
    $\mathit{Pending} \leftarrow \mathit{Pending} \setminus \{\langle m, \_ \rangle\}$
    $\mathit{Delivering} \leftarrow \mathit{Delivering} \cup \{\langle m, ts_f \rangle\}$

procedure DODELIVER ▷ Process q
when: $\exists\, \langle m_i,\, ts_i \rangle \in \mathit{Delivering}$:
    $\forall\, \langle m_j, ts_j \rangle \in (\mathit{Pending} \cup \mathit{Delivering})$:
        $\vee\ m_i \not\sim m_j$
        $\vee\ ts_i < ts_j\ \vee\ (ts_i = ts_j\ \wedge\ m_i < m_j)$
let:
    $G \leftarrow \{\langle m_j,\, ts_j \rangle \in \mathit{Delivering} : \forall\, \langle m_k,\, ts_k \rangle \in \mathit{Delivering} \cup \mathit{Pending} : m_j \not\sim m_k\}$
    $D \leftarrow \{\langle m_i,\, ts_i \rangle\} \cup G$
    $\mathit{Delivering} \leftarrow \mathit{Delivering} \setminus D$
    $\mathit{Delivered} \leftarrow \mathit{Delivered} \cup D$
    for all $\langle m, \_ \rangle \in D$ **do**
        gm-Delivered m

Each process that participates in the algorithm is aware of the same conflict relationship, which changes with the application but is opaque to the algorithm. Participants maintain the following state:

- K is the process' logical clock used to assign a timestamp to each message;
- *PreviousMsgs* is a set used together with the conflict relation to identify conflicting messages;
- *Pending* is a set that holds messages that have been assigned a tentative timestamp;
- *Delivering* is the set of messages with a final timestamp assigned and, therefore, ready to be delivered;
- *Delivered* is the set of delivered messages.

During message exchanges, we use symbols in the tuple to identify which procedure to execute, which closely relates to the processing stage of the message. These symbols are:

- `S0`: no timestamp associated yet;
- `S1`: has a tentative timestamp;
- `S2`: has a final timestamp.

The algorithm starts on the invocation of $\boxed{\text{gm-Send } m \text{ to } \mathcal{G}}$, where the initiator process $p \in \Pi$ will $\boxed{\text{Send } \langle S0,\, m\rangle}$, $\forall\, q \in \mathcal{G}$. To simplify the presentation, we let $m.d = \mathcal{G}$ stand for the destination of message m throughout the algorithm. We have two point-of-views, process p is the message m coordinator, and process q is a process in $m.d$.

Each process $q \in \mathcal{G}$ that $\boxed{\text{Received } \langle S0,\, m\rangle \text{ from } p}$ verifies if there exists a message in the *PreviousMsgs* that conflicts with m using the process conflict relation; if a conflict exists, the process clock will increase by 1 and clear the *PreviousMsgs*. Then, process q associates the current clock value to be the timestamp ts, insert m to the *PreviousMsgs* set and $\langle m,\, ts\rangle$ to the *Pending* set, and $\boxed{\text{Send } \langle S1,\, m,\, ts\rangle \text{ to } p}$.

The coordinator of m, p, executes the next step, responsible for defining the message's final timestamp when it $\boxed{\text{Received } \langle S1,\, m,\, ts\rangle \text{ from } q}$, $\forall\, q \in \mathcal{G}$, that is, when p receives a timestamp proposal from all participants in $\mathcal{G}$. The definitive timestamp ts_f is the maximum ts received from all $q \in \mathcal{G}$. Then, process p $\boxed{\text{Send } \langle S2,\, m,\, ts_f\rangle \text{ to } q}$, $\forall\, q \in \mathcal{G}$.

The next step happens when process $q \in \mathcal{G}$ $\boxed{\text{Received } \langle S2,\, m,\, ts_f\rangle \text{ from } p}$ and $\langle m,\, \_\rangle \in$ *Pending*. Process q clock needs to leap if it is smaller than ts_f. If there exists a message in *PreviousMsgs* that conflicts with m, q's clock is updated to $ts_f + 1$ and *PreviousMsgs* set is cleared; otherwise, when no conflict exists, q's clock leaps to ts_f. The next step is to remove $\langle m,\, \_\rangle$ from the *Pending* and add $\langle m,\, ts_f\rangle$ to the *Delivering* set.

The final step is where processes deliver messages. A process can execute the procedure when there exists a message m in the *Delivering* set that, compared with all other messages in *Delivering* and *Pending*, m is either strictly smaller or does not conflict. A single execution does not deliver only message m; it collects all non-conflicting messages in the *Delivering* set into batch D. Then, for all $m \in D$, the process $\boxed{\text{gm-Delivered } m}$, removes $\langle m, \_\rangle$ from the *Delivering* set, and adds m to the *Delivered*. Observe that no order need to be enforced by this loop and that the algorithm could deliver the batch D all at once and let the application decide the order of processing.

4.1.1 A Little TLC

Algorithm 4.1 is a modified version of ANTUNES' algorithm, resulting from correcting the problems we found after specifying it in TLA$^+$ and verifying it using TLC. We show a condensed version of the specification in Section 4.1.2 and the complete one in Appendix A.3. Although we use formal specifications to find the problems, we found it is easier to describe them and corresponding fixes in the pseudo-code.

How to count

We found problems in procedure `assignSeqNumber` of the original algorithm that violates the Partial Order property. This violation is reproducible in an environment with at least two processes and a pair of conflicting messages. Table 1 shows the algorithm timeline. The tuple $\langle id, ts\rangle$ represents a message, the id guarantees the strict total order, the first line is process p_1 and the second p_2, and $m_1 = \langle 1, \_\rangle$ and $m_2 = \langle 2, \_\rangle$.

In this counter-example, process p_1 receives both messages, while p_2 only m_2. The algorithm proceeds, and eventually, p_2 delivers message m_2. The delayed message m_1 finally arrives at p_2, which does not have conflicting messages, so it proposes a timestamp of 1. Process p_1 has both m_1 and m_2 with the same timestamp, then it uses the messages' strict ordering to sort them, but since p_2 delivery of m_1 was delayed, it can not do the same. This sum of events leads to the Partial Order violation, where process p_1 delivers messages in order m_1 and m_2, and p_2, m_2 and m_1. Algorithm 4.2 is the original version, and Algorithm 4.3 has the fixes applied.

	K	Pending	Delivering	Delivered	PrevMsgs	Network
p_1	0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_2	0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_1	0	$\{\langle 1,0\rangle\}$	$\{\}$	$\{\}$	$\{\langle 1,0\rangle\}$	$\{\langle S1, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_2	0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle,\ \langle 2,1\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S1, \langle 1,0\rangle\rangle,\ \langle S1, \langle 2,1\rangle\rangle\}$
p_2	0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle\ \langle 2,1\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S1, \langle 2,1\rangle\rangle\}$
p_2	0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle,\ \langle 2,1\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	0	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S0, \langle 2,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle,\ \langle 2,1\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S1, \langle 2,0\rangle\rangle\}$
p_2	0	$\{\langle 2,0\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S0, \langle 1,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle,\ \langle 2,1\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S2, \langle 2,1\rangle\rangle\}$
p_2	0	$\{\langle 2,0\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S2, \langle 2,1\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	0	$\{\langle 2,0\rangle\}$	$\{\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S0, \langle 1,0\rangle\rangle,\ \langle S2, \langle 2,1\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\}$	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\langle S0, \langle 1,0\rangle\rangle\}$
p_1	1	$\{\langle 1,0\rangle\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S1,\ \langle 1,1\rangle\rangle\}$
p_2	1	$\{\langle 1,1\rangle\}$	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\}$
p_1	1	$\{\langle 1,0\rangle\}$	$\{\langle 2,1\rangle\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\langle S2,\ \langle 1,1\rangle\}$
p_2	1	$\{\langle 1,1\rangle\}$	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\langle S2,\ \langle 1,1\rangle\}$
p_1	1	$\{\}$	$\{\langle 2,1\rangle, \langle 1,1\rangle\}$	$\{\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\langle 1,1\rangle\}$	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\langle S2,\ \langle 1,1\rangle\}$
p_1	1	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,1\rangle\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\langle 1,1\rangle\}$	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\langle S2,\ \langle 1,1\rangle\}$
p_1	1	$\{\}$	$\{\}$	$\{\langle 1,1\rangle,\ \langle 2,1\rangle\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\langle 1,1\rangle\}$	$\{\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\langle S2,\ \langle 1,1\rangle\}$
p_1	1	$\{\}$	$\{\}$	$\{\langle 1,1\rangle,\ \langle 2,1\rangle\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\}$	$\{\langle 1,1\rangle\}$	$\{\langle 2,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\}$
p_1	1	$\{\}$	$\{\}$	$\{\langle 1,1\rangle,\ \langle 2,1\rangle\}$	$\{\langle 2,0\rangle\}$	$\{\}$
p_2	1	$\{\}$	$\{\}$	$\{\langle 2,1\rangle,\ \langle 1,1\rangle\}$	$\{\langle 1,0\rangle\}$	$\{\}$

Table 1 – Timeline of Partial Order property violation.

Algorithm 4.2 ANTUNES's proposal for the `assignSeqNumber` step.

procedure ASSIGNSEQNUMBER
  when: Received $\langle S2, m, ts_f\rangle$ from p
    $\wedge\ \langle m, \_\rangle \in \mathit{Pending}$
  if $ts_f > K$ **then**
    $K \leftarrow ts_f$
    $\mathit{PreviousMsgs} \leftarrow \emptyset$
  $\mathit{Pending} \leftarrow \mathit{Pending} \setminus \{\langle m, \_\rangle\}$
  $\mathit{Delivering} \leftarrow \mathit{Delivering} \cup \{\langle m, ts_f\rangle\}$

Algorithm 4.3 Changed version for step `assignSeqNumber`.

procedure ASSIGNSEQNUMBER
  when: Received $\langle S2, m, ts_f\rangle$ from p
    $\wedge\ \langle m, \_\rangle \in \mathit{Pending}$
  if $ts_f > K$ **then**
    if $\exists m_i \in \mathit{PreviousMsgs} : m \sim m_i$ **then**
      $K \leftarrow ts_f + 1$
      $\mathit{PreviousMsgs} \leftarrow \emptyset$
    else
      $K \leftarrow ts_f$
  $\mathit{Pending} \leftarrow \mathit{Pending} \setminus \{\langle m, \_\rangle\}$
  $\mathit{Delivering} \leftarrow \mathit{Delivering} \cup \{\langle m, ts_f\rangle\}$

We change procedure `assignSeqNumber` to verify if a conflict exists instead of only leaping the clock. This approach creates a bond between the process clock, *PreviousMsgs* set, and the conflict relation. Every time we clear the *PreviousMsgs* set, we also increase the clock. We could solve this problem by increasing the clock at every received message, following Lamport's work (LAMPORT, 2019), but this is not the behavior we want. We want to keep the timestamps as low as possible, increasing the local clock only when messages conflict (ANTUNES, 2019).

We change the procedure `doDeliver`. The original proposition relied on the *Delivering* set being synchronized and ordered during the procedure execution, using the function *NCSF* (Non-conflicting set function), which is not present in the algorithm, to return the messages ready to deliver, in the correct order, without violating the properties. We use a predicate that verifies that there exists a message that commutes with all others or is strictly smaller than all others, a more direct verification without violating the properties. Lamport clock uses a similar predicate to create a total order of the system's events (LAMPORT, 2019). Algorithm 4.4 is the original algorithm, and Algorithm 4.5 is our proposal.

Algorithm 4.4 Original `doDeliver` by ANTUNES.

procedure DODELIVER
  let: $CandidateSet = \{\langle m_i, ts_i \rangle :$
    $\langle m_i, ts_i \rangle \in Delivering :$
      $\forall \langle m_j, ts_j \rangle \in (Pending \cup Delivering) :$
        $ts_i \leq ts_j \}$

  when: $CandidateSet \neq \emptyset$
    $D = NCSF(CandidateSet)$
    $Delivering \leftarrow Delivering \setminus D$
    $Delivered \leftarrow Delivered \cup D$
    for all $\langle m_i, \_ \rangle \in D$ **do**
      gm-Delivered $\langle m \rangle$

Algorithm 4.5 Changed `doDeliver`.

procedure DODELIVER
  when: $\exists \langle m_i, ts_i \rangle \in Delivering :$
    $\forall \langle m_j, ts_j \rangle \in (Pending \cup Delivering) :$
      $\vee\ m_i \not\sim m_j$
      $\vee\ ts_i < ts_j \vee (ts_i = ts_j \wedge m_i < m_j)$
  let:
    $G \leftarrow \{\langle m_j, ts_j \rangle \in Delivering :$
      $\forall \langle m_k, ts_k \rangle \in Delivering \cup Pending :$
        $m_j \not\sim m_k\}$
    $D \leftarrow \{\langle m_i, ts_i \rangle\} \cup G$
    $Delivering \leftarrow Delivering \setminus D$
    $Delivered \leftarrow Delivered \cup D$
    for all $\langle m, \_ \rangle \in D$ **do**
      gm-Delivered $\langle m \rangle$

Learning to count again

Our algorithm in Algorithm 4.1 solves the Generic Multicast problem without violating the properties. To check that the properties written in TLA$^+$ are correct, we manually introduce bugs to cause a violation, and the model checker must report the error. We did this on all the properties, and during the Partial Order property, we noticed something strange. Remembering that, the Partial Order property guarantees that processes that deliver a pair of conflicting messages do so in the same order.

In TLA$^+$, to capture the order a process delivers a message m, we use the *Delivered* set, where we insert the tuple $\langle |Delivered|, m \rangle$. This avoids an additional variable in the algorithm, and it is easier to do operations over a set in TLA$^+$. The bug we introduced was to use the tuple $\langle 0, m \rangle$, pretending that processes delivered all the messages in a single batch. To our surprise, this does not violate that Partial Order property.

The property on page 31 has that process p_1 delivers the conflicting messages m_1

before m_2 if, and only if, process p_2 does the same. Augmenting each process p in Π with a sequence $\mathcal{I}_p$, that once a message m is $\boxed{\text{gm-Delivered } m}$ by p, it adds m to the end of $\mathcal{I}_p$. For a message m and process p in Π, $Idx(m,\, p)$ returns the position of m on p's sequence. We define L to be true when processes p_1 and p_2 in Π both delivered m_1 and m_2, and $m1 \sim m2$. First, to ease the formulae writing, we define:

$$p = Idx(m_1, p_1) < Idx(m_2, p_1) \quad q = Idx(m_1, p_2) < Idx(m_2, p_2)$$

We have that Partial Order (PO):

$$R = p \iff q \tag{8a}$$
$$\text{PO} = L \implies R \tag{8b}$$

We can rewrite Equation (8a) as:

$$\begin{aligned} R &= (p \implies q) \wedge (q \implies p) \\ &= (\neg p \vee q) \wedge (\neg q \vee p) \end{aligned} \tag{8c}$$

In our example, we delivered all messages in the same position, that is, $\forall p_i \in \Pi$ and $\forall m_i, m_j \in \mathcal{M}$, we have that $Idx(m_i, p_i) = Idx(m_j, p_i)$. We have that both p and q are false. Substituting the values in Equation (8c), we have that:

$$\begin{aligned} R &= (\neg\text{false} \vee \text{false}) \wedge (\neg\text{false} \vee \text{false}) \\ &= (\text{true} \vee \text{false}) \wedge (\text{true} \vee \text{false}) \\ &= \text{true} \wedge \text{true} \end{aligned}$$

In this case, we have that Equation (8a) is always true. Substituting in Equation (8b), L implies in something true, evaluating everything to true. That is, if processes (somehow) deliver multiple conflicting messages in a single operation, it does not violate the Partial Order property.

To strengthen the Partial Order property, we introduce an additional property named *Collision*. Informally, this property requires that, given a pair of conflicting messages, a process must deliver them in some order. We define the Collision property as:

- **Collision**: If a process $p \in \Pi$, $\boxed{\text{gm-Delivered } m_i}$ and $\boxed{\text{gm-Delivered } m_j}$, and $m_i \sim m_j$, then p $\boxed{\text{gm-Delivered } m_i}$ before $\boxed{\text{gm-Delivered } m_j}$ or p $\boxed{\text{gm-Delivered } m_j}$ before $\boxed{\text{gm-Delivered } m_i}$.

An algorithm with Collision and Partial Order properties guarantees that messages have a correct order based on the conflict relation. The Collision property is more of a theoretical reinforcement. On an actual execution, messages are delivered one at a time; messages will have an order, there is no way to violate the property.

Observe that we achieve the desired effect of delivering the conflicting messages in order with the "happened before" relation (LAMPORT, 2019). That is, following the definition of $\rightarrow$ (LAMPORT, 2019), then we can write the Partial Order property as:

- **Partial Order**: if processes p_1, $p_2 \in \Pi$ both $\boxed{\text{gm-Delivered } m_1}$ and $\boxed{\text{gm-Delivered } m_2}$, and $m_1 \sim m_2$, then p_1 $\boxed{\text{gm-Delivered } m_1} \rightarrow \boxed{\text{gm-Delivered } m_2}$, if, and only if, p_2 $\boxed{\text{gm-Delivered } m_1} \rightarrow \boxed{\text{gm-Delivered } m_2}$.

The final form

The Generic Multicast 0 specification is now complete. We fixed a problem that violates the Partial Order property on procedure `assignSeqNumber` and changed procedure `doDeliver` with a simpler predicate. There exists room for improvement. Currently, messages have a static destination, whereas it would be more interesting if we checked every destination possible.

We also had a theoretical discussion about delivering messages in a single batch, where we came up with an additional property. The Collision property requires that a process order conflicting messages, albeit the property might be unnecessary in a real case. We added a check for the Collision in our specifications.

From our first experience with TLA$^+$, we found an intricate problem that needed a specific combination of events to trigger a violation. Such a problem is hard to find by only reasoning over the algorithm, which would be much harder to find without TLA$^+$ and TLC.

4.1.2 Generic Multicast 0 in TLA$^+$

The Generic Multicast 0 specification was the first developed in the present work. We refined the specification in multiple steps until its final form, around 300 lines, including comments and helper procedures. Next, we review portions of the specification that roughly correspond to Algorithm 4.1 on page 50.

In the beginning

During the specification's model checking, we vary the number of messages, processes, and conflict relation to check different scenarios. To simplify this arrangement, we externalized these settings as constants: `NPROCESSES`, denoting the number of processes the model will simulate; `INITIAL_MESSAGES`, a finite set with the messages to initialize the algorithm; and `CONFLICTR`, the conflict relation the algorithm requires. These constants do not appear in the TLA$^+$ specification we display here. Algorithm 4.6 and Algorithm 4.7 have the TLA$^+$ representations.

Algorithm 4.6 Generic Multicast 0 in TLA$^+$– Part 1.

$$
\begin{array}{l}
\mathit{AssignTimestamp}(\mathit{self}) \triangleq \\
\quad \land\ \mathit{QuasiReliable}!\mathit{Receive}(\mathit{self}, 1, \\
\qquad \textsc{lambda}\ t : \\
\qquad\quad \land\ t[1] = \text{“S0”} \\
\qquad\quad \land\ \mathit{AssignTimestampHandler}(\mathit{self}, t[2])) \\
\\
\textsc{local}\ \mathit{AssignTimestampHandler}(\mathit{self}, \mathit{msg}) \triangleq \\
\quad \land\ \lor\ \land\ \mathit{HasConflict}(\mathit{self}, \mathit{msg}) \\
\qquad\quad \land\ K' = [K\ \textsc{except}\ ![\mathit{self}] = K[\mathit{self}] + 1] \\
\qquad\quad \land\ \mathit{PreviousMsgs}' = [\mathit{PreviousMsgs}\ \textsc{except}\ ![\mathit{self}] = \{\mathit{msg}\}] \\
\qquad \lor\ \land\ \neg \mathit{HasConflict}(\mathit{self}, \mathit{msg}) \\
\qquad\quad \land\ K' = [K\ \textsc{except}\ ![\mathit{self}] = K[\mathit{self}]] \\
\qquad\quad \land\ \mathit{PreviousMsgs}' = \\
\qquad\qquad [\mathit{PreviousMsgs}\ \textsc{except}\ ![\mathit{self}] = \mathit{PreviousMsgs}[\mathit{self}] \cup \{\mathit{msg}\}] \\
\quad \land\ \mathit{Pending}' = [\mathit{Pending}\ \textsc{except}\ ![\mathit{self}] = \mathit{Pending}[\mathit{self}] \cup \{\langle K'[\mathit{self}], \mathit{msg}\rangle\}] \\
\quad \land\ \mathit{QuasiReliable}!\mathit{SendMap}(\textsc{lambda}\ \mathit{dest}, S : \\
\qquad \mathit{SendOriginatorAndRemoveLocal}(\mathit{self}, \mathit{dest}, \\
\qquad\ \ \langle \text{“S1”}, K'[\mathit{self}], \mathit{msg}, \mathit{self}\rangle, \langle \text{“S0”}, \mathit{msg}\rangle, S)) \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{Delivering}, \mathit{Delivered}, \mathit{Votes}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{ComputeSeqNumber}(\mathit{self}) \triangleq \\
\quad \land\ \mathit{QuasiReliable}!\mathit{Receive}(\mathit{self}, 1, \\
\qquad \textsc{lambda}\ t : \\
\qquad\quad \land\ t[1] = \text{“S1”} \\
\qquad\quad \land\ t[3].o = \mathit{self} \\
\qquad\quad \land\ \mathit{ComputeSeqNumberHandler}(\mathit{self}, t[2], t[3], t[4])) \\
\\
\textsc{local}\ \mathit{ComputeSeqNumberHandler}(\mathit{self}, \mathit{ts}, \mathit{msg}, \mathit{origin}) \triangleq \\
\quad \land\ \textsc{let} \\
\qquad \mathit{vote} \triangleq \langle \mathit{msg.id}, \mathit{origin}, \mathit{ts}\rangle \\
\qquad \mathit{election} \triangleq \{v \in (\mathit{Votes}[\mathit{self}] \cup \{\mathit{vote}\}) : v[1] = \mathit{msg.id}\} \\
\qquad \mathit{elected} \triangleq \mathit{Max}(\{x[3] : x \in \mathit{election}\}) \\
\quad\ \ \textsc{in} \\
\qquad \land\ \lor\ \land\ \mathit{Cardinality}(\mathit{election}) = \mathit{Cardinality}(\mathit{msg.d}) \\
\qquad\qquad \land\ \mathit{Votes}' = [\mathit{Votes}\ \textsc{except}\ ![\mathit{self}] = \{x \in \mathit{Votes}[\mathit{self}] : x[1] \neq \mathit{msg.id}\}] \\
\qquad\qquad \land\ \mathit{QuasiReliable}!\mathit{SendMap}(\textsc{lambda}\ \mathit{dest}, S : \\
\qquad\qquad\qquad (S \setminus \{\langle \text{“S1”}, \mathit{ts}, \mathit{msg}\rangle\}) \cup \{\langle \text{“S2”}, \mathit{elected}, \mathit{msg}\rangle\}) \\
\qquad\quad \lor\ \land\ \mathit{Cardinality}(\mathit{election}) < \mathit{Cardinality}(\mathit{msg.d}) \\
\qquad\qquad \land\ \mathit{Votes}' = [\mathit{Votes}\ \textsc{except}\ ![\mathit{self}] = \mathit{Votes}[\mathit{self}] \cup \{\mathit{vote}\}] \\
\qquad\qquad \land\ \mathit{QuasiReliable}!\mathit{Consume}(1, \mathit{self}, \langle \text{“S1”}, \mathit{ts}, \mathit{msg}, \mathit{origin}\rangle) \\
\qquad \land\ \textsc{unchanged}\ \langle K, \mathit{PreviousMsgs}, \mathit{Pending}, \mathit{Delivering}, \mathit{Delivered}\rangle
\end{array}
$$

Algorithm 4.7 Generic Multicast 0 in TLA+– Part 2.

```
AssignSeqNumber(self) ≜
    ∧ QuasiReliable!ReceiveAndConsume(self, 1,
        LAMBDA t_1 :
            ∧ t_1[1] = "S2"
            ∧ ∃ t_2 ∈ Pending[self] : t_1[3].id = t_2[2].id
                ∧ AssignSeqNumberHandler(self, t_1[2], t_1[3])
                ∧ Pending' = [Pending EXCEPT ![self] = @ \ {t_2}])

LOCAL AssignSeqNumberHandler(self, ts, msg) ≜
    ∧ ∨ ∧ ts > K[self]
        ∧ ∨ ∧ HasConflict(self, msg)
            ∧ K' = [K EXCEPT ![self] = ts + 1]
            ∧ PreviousMsgs' = [PreviousMsgs EXCEPT ![self] = {}]
          ∨ ∧ ¬HasConflict(self, msg)
            ∧ K' = [K EXCEPT ![self] = ts]
            ∧ UNCHANGED PreviousMsgs
      ∨ ∧ ts ≤ K[self]
        ∧ UNCHANGED ⟨K, PreviousMsgs⟩
    ∧ Delivering' = [Delivering EXCEPT ![self] = Delivering[self] ∪ {⟨ts, msg⟩}]
    ∧ UNCHANGED ⟨Votes, Delivered⟩
```

```
DoDeliver(self) ≜
    ∃ ⟨ts_1, m_1⟩ ∈ Delivering[self] :
        ∧ ∀ ⟨ts_2, m_2⟩ ∈ (Delivering[self] ∪ Pending[self]) \ {⟨ts_1, m_1⟩} :
            ∨ ¬CONFLICTR(m_1, m_2)
            ∨ ts_1 < ts_2 ∨ (m_1.id < m_2.id ∧ ts_1 = ts_2)
        ∧ LET
            T ≜ Delivering[self] ∪ Pending[self]
            G ≜ {t_i ∈ Delivering[self] : ∀ t_j ∈ T \ {t_i} :
                    ¬CONFLICTR(t_i[2], t_j[2])}
            D ≜ {m_1} ∪ {t[2] : t ∈ G}
          IN
            ∧ Delivering' = [Delivering EXCEPT ![self] = @ \ (G ∪ {⟨ts_1, m_1⟩})]
            ∧ Delivered' = [Delivered EXCEPT ![self] =
                    Delivered[self] ∪ Enumerate(Cardinality(Delivered[self]), D)]
            ∧ UNCHANGED ⟨QuasiReliableChannel, Votes, Pending, PreviousMsgs, K⟩
```

Thou shalt gm-Send

We use a record (LAMPORT, 2002) to represent the messages, written as [key $\mapsto$ value], each with a unique identifier that guarantees the strict total order, the destination, and the originator process.

To simulate multiple processes, we use a record, mapping the process identifier to the variables. For example, the *Pending* set starts as $Pending = [i \in Processes \mapsto \{\}]$. Each procedure has the process identifier as an argument, which we use to access the corresponding variables.

We abstract the channels that connect processes using a set, guaranteeing the channel's quasi-reliable properties. The participants send and receive messages by adding or removing elements, with no delivery order, message loss, duplication, or spontaneous creation. We use the contents of the `INITIAL_MESSAGES` constant to initialize the network, meaning that the procedure in line 7 of Algorithm 4.1 does not exist in the specification. Following this approach was an easier way to introduce a finite number of messages in the algorithm. A drawback is that the model checker does not try all possibilities for the destination and initiator process, varying the multicast behavior.

4.2 Generic Multicast 1

The second algorithm we verify with TLA$^+$ is Generic Multicast 1, an algorithm based on FRITZKE et al.'s algorithm with improvements from SCHIPER; PEDONE. Although this model does not fit a real-world production environment (ANTUNES, 2019), it is interesting to introduce the algorithm in an environment where failures exist. The algorithm uses a replication approach, dealing with a group of processes instead of a single process, where groups are reliable (ANTUNES, 2019). A group can represent one site, wherein members rely on the local site link for communication (ANTUNES, 2019).

It works similarly to Generic Multicast 0, ordering messages by the timestamp and, in the process, using the conflict relation, but Generic Multicast 1 does not use a coordinator to decide a final timestamp. This algorithm assumes an asynchronous system, with crash-stop failures but fault-tolerant partitions, and that an Atomic Broadcast primitive is available. Algorithm 4.8 presents the algorithm pseudo-code, where all procedures have an atomic execution.

As Generic Multicast 0, processes participating in the algorithm are aware of the same conflict relation. Besides all the symbols `S0`, `S1`, and `S2`, this algorithm uses an additional one, `S3`, meaning that a message has a final timestamp, is ready to be delivered, and the local group is synchronized (ANTUNES, 2019). Each process has the following state:

- K is the process' logical clock;
- *PreviousMsgs* is the set used together with the conflict relation to identify conflicting messages;
- *Mem* is a memory structure that holds the messages we are processing without ever creating duplicated entries for a message.

We follow the execution from process p's point-of-view, where p is a correct process. The algorithm starts on the invocation of $\boxed{\text{gm-Send } m \text{ to } \mathcal{G}}$, and we define $m.d = \mathcal{G}$ to simplify the algorithm presentation. The initiator process will $\boxed{\text{ab-Send } \langle m, S0, 0\rangle \text{ to } g}$, $\forall g \in \mathcal{G}$, where each primitive use is independent.

Algorithm 4.8 Generic Multicast 1.

Variables:
$K \leftarrow 0$, $Mem \leftarrow \emptyset$, $PreviousMsgs \leftarrow \emptyset$

procedure GM-SEND(m, $\mathcal{G}$)
  let $m.d = \mathcal{G}$
  for all $g \in \mathcal{G}$ **do**
    ab-Send $\langle m, S0, 0\rangle$ to g ▷ Atomic Broadcast

procedure COMPUTEGROUPSEQNUMBER
when: ab-Delivered $\langle m, S0, ts\rangle$ ▷ Atomic Broadcast Deliver
  if $\exists\ m_i \in PreviousMsgs : m \sim m_i$ **then**
    $K \leftarrow K + 1$
    $PreviousMsgs \leftarrow \emptyset$
  $PreviousMsgs \leftarrow PreviousMsgs \cup \{m\}$
  if $|m.d| > 1$ **then**
    $Mem \Leftarrow \langle m, S1, K\rangle$
    for all $g \in m.d$ **do**
      for all $p \in g$ **do**
        Send $\langle m, S1, K\rangle$ to p
  else
    $Mem \Leftarrow \langle m, S3, K\rangle$

procedure GATHERGROUPSTIMESTAMPS
when: $\exists\ m : \langle m, S1, ts\rangle \in Mem$
  $\wedge\ \forall\ g \in m.d :$
    $\exists\ p \in g :$ Received $\langle m, S1, v\rangle$
  $ts_f \leftarrow \max(\{v :$ Received $\langle m, S1, v\rangle\})$
  if $ts < ts_f$ **then**
    ab-Send $\langle m, S2, ts_f\rangle$ to G_{local} ▷ Local Atomic Broadcast
  $Mem \Leftarrow \langle m, S3, ts_f\rangle$

procedure SYNCHRONIZEGROUP
when: ab-Delivered $\langle m, S2, ts_f\rangle$ ▷ Atomic Broadcast Deliver
  if $ts_f > K$ **then**
    $K \leftarrow ts_f$
    $PreviousMsgs \leftarrow \emptyset$
  if $\exists\ \langle m, S1, \_\rangle \in Mem$ **then**
    $Mem \Leftarrow \langle m, S3, ts_f\rangle$

procedure DODELIVER
when: $\exists\ \langle m_i, S3, ts_i\rangle \in Mem :$
  $\wedge\ \forall\ \langle m_j, \_, ts_j\rangle \in Mem :$
    $\vee\ m_i \nsim m_j$
    $\vee\ ts_i < ts_j$
    $\vee\ \wedge\ ts_i = ts_j$
      $\wedge m_i < m_j$
let:
  $NC \leftarrow \{\langle m_j, S3, \_\rangle \in Mem : \forall\ \langle m_k, \_, \_\rangle \in Mem : m_j \nsim m_k\}$
  $D \leftarrow \{\langle m_i, s_i, ts_i\rangle\} \cup NC$
  $Mem \leftarrow Mem \setminus D$
  for all $\langle m, \_, \_\rangle \in D$ **do**
    gm-Delivered m

Procedure `computeGroupSeqNumber` executes when p receives the tuple $\langle m, S0, ts\rangle$ through the Atomic Broadcast primitive. On receiving $\langle m, S0, ts\rangle$, it is the first time p deals with m, so p verifies if m conflicts with any message in the *PreviousMsgs* set. If a conflict exists, p increases its local clock by 1 and clears the *PreviousMsgs* set. Lastly, p inserts m into the *PreviousMsgs* set for later conflict verifications. Here, the algorithm branches based on the destination, an optimization proposed by SCHIPER; PEDONE.

When a message has a single group in the destination, the process can store the tuple $\langle m,\ S3,\ K\rangle$ in the *Mem* structure. Because m's destination is a single group, and p received it through the Atomic Broadcast, the message is at the desired destination in the correct order. The tuple $\langle m,\ S3,\ K\rangle$ associates m with a final timestamp and the symbol `S3` to identify it as ready to be delivered.

When a message has multiple groups in the destination, the participants must collaborate to agree on the final timestamp. So p proposes a timestamp with its current clock value to every participant in every group using $\boxed{\text{Send } \langle m, S1, K\rangle \text{ to } \_}$ and store the tuple $\langle m, S1, K\rangle$ in *Mem*.

Processes execute the next procedure, `gatherGroupsTimestamps`, to decide a message's final timestamp, a necessary step when multiple groups are in the message's destination. After receiving a vote v with $\boxed{\text{Received } \langle m, S1, v\rangle \text{ from each group}}$ and p has the tuple $\langle m, S1, ts\rangle$ in *Mem*, the selected timestamp ts_f is the maximum vote received. If p's vote in *Mem*, ts, is smaller than the final timestamp ts_f means there exists a group with a higher clock, and the local group needs to synchronize, so p $\boxed{\text{ab-Send } \langle m, S2, ts_f\rangle \text{ to } G_{local}}$. Finally, since the message has a final timestamp ts_f, p inserts the tuple $\langle m, S3, ts\rangle$ to *Mem*.

When deciding the final timestamp, if the local group needs synchronization, p broadcasts the decided timestamp to the local group. Procedure `synchronizeGroup` executes when receiving a message with symbol `S2` through the Atomic Broadcast. Upon receiving the tuple $\langle m, S2, ts_f\rangle$, if the current clock has a value smaller than the ts_f, p leaps the clock to ts_f and clears the *PreviousMsgs* set. If p has m in *Mem* associated with the symbol `S1` means that the synchronization message arrived before all necessary votes, so p can associate m with the symbol `S3`, avoiding the need for gathering all proposals.

The last step is where processes deliver the messages, procedure `doDeliver`. For a tuple $\langle m, s, ts\rangle$, the message m is ready to be delivered when $s =$ `S3`, and, comparing m with all other tuples in *Mem*, either m does not conflict with any other message or the pair $\langle m, ts\rangle$ is the smallest. The process then collects and removes all non-conflicting messages with the symbol `S3` in *Mem* and $\boxed{\text{gm-Delivered } \_}$ one at a time.

4.2.1 Handyman's Mode

The original algorithm had problems. Before starting, this algorithm inherits the changes to the procedure `doDeliver` and the Collision property from Generic Multicast 0. We do not describe these inherited fixes, only focusing on the Generic Multicast 1 problems.

The first problem could lead to an infinite loop broadcasting the message to the local group. Algorithm 4.9 is the original proposition. Procedure `groupABroadcast` does a local broadcast for messages with symbols `S0` or `S2`. This procedure can execute multiple times because the symbol does not update after the broadcast. We solve this problem by removing `groupABroadcast` procedure and using the Atomic Broadcast directly where needed. Also, observe that, in this approach, each process within a group is doing an Atomic Broadcast, and since no filtering exists, it is possible to gm-Delivered _ a message more than once, violating the Integrity property. An approach where we include a filter would be more complicated, distinguish between new messages and ones we delivered before is difficult without keeping a record of all deliveries. To solve this issue, we remove the Reliable Multicast and use the Atomic Broadcast for each group, alleviating the need for an additional primitive for group communication.

Algorithm 4.9 Original Generic Multicast 1 beginning.

procedure GM-SEND(m, $\mathcal{G}$)
  let: $m.d = \mathcal{G}$
  rm-Send $\langle m \rangle$ to $\mathcal{G}$

procedure ENQUEUEMESSAGE
when: rm-Delivered $\langle m \rangle$
  $Mem \Leftarrow \langle m, S0, 0 \rangle$

procedure GROUPABROADCAST
when: $\exists \langle m, s, ts \rangle \in Mem : s \in \{S0, S2\}$
  ab-Send $\langle m, s, ts \rangle$ to G_{local}

The other fix applies to procedure `computeGroupSeqNumber` when processes exchange their proposals. On the original algorithm, a process sends its proposal to processes in $(m.d \setminus G_{local})$, that is, everyone but the ones in the local group. At the same time, procedure `gatherGroupsTimestamps` expects a message from all groups, including the local one, leading to the algorithm never delivering messages. We solve this problem by sending the proposal to every process of every group, where if a process wants to skip sending a message to itself and only invoke a method, we leave it as an implementation detail.

We also did a complete overhaul on procedure `gatherGroupsTimestamps`. The original version is in Algorithm 4.10, and Algorithm 4.11 has our version with specific changes highlighted.

Algorithm 4.10 ANTUNES' proposal for the `gatherGroupsTimestamps` step.

```
procedure GATHERGROUPSTIMESTAMPS
  when: ∀ g ∈ m.d :
          ∃ p ∈ g : Received⟨m, S1, v⟩
    ts_f ← max({v : Received⟨m, S1, v⟩})
    if ts ≥ ts_f then
        Mem ⇐ ⟨m, S3, ts_f⟩
    else
        Mem ⇐ ⟨m, S2, ts_f⟩
        ab-Send ⟨m, S2, ts_f⟩ to G_local
```

Algorithm 4.11 Our version for procedure `gatherGroupsTimestamps`.

```
procedure GATHERGROUPSTIMESTAMPS
  when: ∃ m : [⟨m, S1, ts⟩ ∈ Mem] ¹
          ∧ ∀ g ∈ m.d :
            ∃ p ∈ g : Received⟨m, S1, v⟩
    ts_f ← max({v : Received⟨m, S1, v⟩})
    if ts < ts_f then
        ab-Send ⟨m, S2, ts_f⟩ to G_local
    [Mem ⇐ ⟨m, S3, ts_f⟩] ²
```

Starting with (1) on Algorithm 4.11, a subtle problem. This procedure executes after receiving a timestamp proposal from at least one process of all groups in the message's destination. Without assumptions about process speed and message delay, a process could receive all the proposals necessary to proceed before receiving the message through the Atomic Broadcast. This behavior could lead to a message locking itself off deliver by "going back in time" or the associated symbol, first executing procedure `gatherGroupsTimestamps` and then executing procedure `computeGroupSeqNumber`. To solve this, we strengthen the `gatherGroupsTimestamps` predicate by requiring the message to be in *Mem* with the symbol `S1`. Note that this is a requirement for execution, where the process must collect the message's votes received in the meantime.

With (2), we solve the possibility of delivering messages multiple times. When the group has a clock with a smaller value than the message's decided timestamp, it must synchronize by doing an Atomic Broadcast and leaping the clock to the timestamp value. Originally, the algorithm inserted and then broadcasted the tuple $\langle m, S2, ts_f \rangle$, and when received, the process would associate m with `S3` without verifying if m exists, which could lead to multiple deliveries. To handle this problem, once m has the decided timestamp, we can insert the message in *Mem* associated with the symbol `S3`, marking it as deliverable. To synchronize the group, we extracted the method that handles the symbol `S2` into its own procedure, `synchronizeGroup`, making it easier to read and, most importantly, idempotent; it can receive the same message multiple times without causing the message to be delivered multiple times. We also insert a shortcut: if the process receives the synchronization message m before receiving all necessary proposals and has sent its timestamp proposal for m to the others, it can mark m as ready for delivery. The shortcut is a way to avoid more participants executing an unnecessary local Atomic Broadcast.

4.2.2 Handling Incorrectness

Generic Multicast 1 algorithm works in an environment with incorrect processes, having a crash-stop model. With an incorrect initiator, the message may or may not be delivered. The deliver is atomic, that is, either everyone in the destination delivers, or none does. When a participant fails, it will never deliver any message afterwards. There are two points of attention in the algorithm, (i) at the beginning when executing multiple Atomic Broadcast and (ii) when using the Atomic Broadcast to synchronize the group.

At (i), if the initiator is incorrect, the broadcasted message may or may never be delivered. In cases where a message is delivered only partially to a subset of the total destination, it is not possible to decide on the message's final timestamp, and therefore it is impossible to deliver the message. Observe that in such a scenario, the failed message lingers in the processes' *Mem* structure forever, without ever making progress. There is room for improvement, a way that groups could aid each other when failures occur or add a mechanism to clear the messages that do not make progress.

And for (ii), where one could think of optimizing to only one process to do an Atomic Broadcast for synchronization. We did not try this because the process could be incorrect, so all participants can execute the steps for deciding the timestamp. The procedure that handles the synchronization message is idempotent to tolerate duplicated messages, where the clock synchronizes only once and can leap to ready for delivery only once. After one process within the group succeeds in broadcasting the synchronization, it may not be necessary for the others participants to do it too. Since the group is reliable, there will be a successful broadcast.

Processes that fail in other points of the algorithm cause no harm. Crashing before sending a proposal is not a problem because the group is reliable, so at least one participant sends the proposal on the group's behalf. Failing at other points only leads to that participant not delivering the message because it stops forever.

4.2.3 Fault-Tolerant Specification

The TLA$^+$ specification for this algorithm was more complex to develop, even though the resulting TLA$^+$ specification is slightly smaller when compared to Generic Multicast 0. The reason for the specification to be smaller is the modularization employed. That is, specification splits into modules, where the network primitives for process communication, the Atomic Broadcast, and the *Mem* structure are separate modules. This modularization helps the abstraction, keeping the core algorithm and increasing reusability.

The specification for the algorithm itself is in Appendix A.4. The specification also contains some required constants provided to the model checker. The `INITIAL_MESSAGES` and `CONFLICTR` are from the previous specification. A new variable introduced is the `NGROUPS`, which specifies the number of groups the model will simulate, and `NPROCESSES` specifies the number of processes each group has. The abstraction for communication between processes and *Mem* uses a set, and the Atomic Broadcast uses a sequence. These TLA$^+$ modules are available in Appendix A.1.

In this specification, we model more communication primitives, groups of processes, and data structures. This combination leads to too many states for the model checker to verify. We could not run the model checker for larger models because of the completion time and disk usage. Even during development, smaller models could take minutes to complete. Appendix A.6.2 includes the runs and configurations we checked.

The current specification has room for improvement. This specification has the same problem with the static messages' initialization as Generic Multicast 0. The specification does not model process failure, as that would increase the number of states even more, but since groups are reliable, this should not be a problem and would be an improvement for completeness' sake.

4.3 Generic Multicast 2

The third and last algorithm we specified is Generic Multicast 2 (ANTUNES, 2019). This algorithm is a direct extension of Generic Multicast 1, which replaces the Atomic Broadcast primitive with Generic Broadcast; we call this form *truly-generic* (ANTUNES, 2019). This replacement is in line with our goal of only ordering messages that require ordering (ANTUNES, 2019); if ordering adds a cost to an algorithm, we should try to keep this cost as low as possible (PEDONE; SCHIPER, 1999). Note that the synchronization messages, that is, the ones that are Generic Broadcast with symbol `S2`, these messages must conflict with each order (ANTUNES, 2019). A truly-generic nature also minimizes the convoy effect compared to the Atomic Broadcast version (ANTUNES, 2019). Algorithm 4.12 shows the Generic Multicast 2 pseudo-code. This algorithm works as the Generic Multicast 1, so we do not repeat ourselves in the explanation.

This extension also means that this algorithm inherits all the fixes. The fixes include the loop broadcasting the message to the local group, the proposals exchanges not including the local group, and a message locked out of delivery or delivering multiple times. We remove the Reliable Multicast use and add procedure `synchronizeGroup`.

During the Generic Multicast 1 specification, we invest some effort into modularization, decoupling the group communication abstraction. This investment pays here. We only needed to abstract the Generic Broadcast in its module and use it in the specification. We use a sequence of sets to abstract the Generic Broadcast and use the same conflict relation that the Generic Multicast uses. Appendix A.6.3 contains the TLA$^+$ specification.

This specification is as cumbersome as Generic Multicast 1. We did not run the model checker for larger models, which could take too much time and storage space. The Generic Broadcast abstraction increases the number of states. Appendix A.6.3 has the configurations and checks we did. And as this specification inherits everything from Generic Multicast 1, it also inherits all the issues, which are the static messages' initialization, the number of states during model checking, and not model incorrect processes.

Algorithm 4.12 Generic Multicast 2.

Variables:
$K \leftarrow 0$, $Mem \leftarrow \emptyset$, $PreviousMsgs \leftarrow \emptyset$

procedure GM-SEND(m, $\mathcal{G}$)
  let $m.d = \mathcal{G}$
  for all $g \in \mathcal{G}$ **do**
    gb-Send $\langle m,\ S0,\ 0\rangle$ to g ▷ Generic Broadcast

procedure COMPUTEGROUPSEQNUMBER
when: gb-Delivered $\langle m,\ S0,\ ts\rangle$ ▷ Generic Broadcast Deliver
  if $\exists\ m_i \in PreviousMsgs : m \sim m_i$ **then**
    $K \leftarrow K + 1$
    $PreviousMsgs \leftarrow \emptyset$
  $PreviousMsgs \leftarrow PreviousMsgs \cup \{m\}$
  if $|m.d| > 1$ **then**
    $Mem \Leftarrow \langle m, S1, K\rangle$
    for all $g \in m.d$ **do**
      for all $p \in g$ **do**
        Send $\langle m, S1, K\rangle$ to p
  else
    $Mem \Leftarrow \langle m, S3, K\rangle$

procedure GATHERGROUPSTIMESTAMPS
when: $\exists\ m : \langle m, S1, ts\rangle \in Mem$
    $\wedge\ \forall\ g \in m.d :$
      $\exists\ p \in g :$ Received $\langle m, S1, v\rangle$
  $ts_f \leftarrow \max(\{v :$ Received $\langle m, S1, v\rangle\})$
  if $ts < ts_f$ **then**
    gb-Send $\langle m, S2, ts_f\rangle$ to G_{local} ▷ Local Generic Broadcast
  $Mem \Leftarrow \langle m, S3, ts_f\rangle$

procedure SYNCHRONIZEGROUP
when: gb-Delivered $\langle m,\ S2,\ ts_f\rangle$ ▷ Generic Broadcast Deliver
  if $ts_f > K$ **then**
    $K \leftarrow ts_f$
    $PreviousMsgs \leftarrow \emptyset$
  if $\exists\ \langle m, S1, \_\rangle \in Mem$ **then**
    $Mem \Leftarrow \langle m, S3, ts_f\rangle$

procedure DODELIVER
when: $\exists\ \langle m_i, S3, ts_i\rangle \in Mem :$
  $\wedge\ \forall\ \langle m_j, \_, ts_j\rangle \in Mem :$
    $\vee\ m_i \nsim m_j$
    $\vee\ ts_i < ts_j$
    $\vee \wedge\ ts_i = ts_j$
      $\wedge m_i < m_j$
let:
  $NC \leftarrow \{\langle m_j,\ S3,\ \_\rangle \in Mem : \forall\ \langle m_k,\ \_,\ \_\rangle \in Mem : m_j \nsim m_k\}$
  $D \leftarrow \{\langle m_i, s_i, ts_i\rangle\} \cup NC$
  $Mem \leftarrow Mem\ \setminus\ D$
  for all $\langle m,\ \_,\ \_\rangle \in D$ **do**
    gm-Delivered m

4.4 Specifying with TLA$^+$

We formalized and verified all of ANTUNES' algorithms in TLA$^+$ and checked with TLC. We developed our specifications in a refinement process, building a single block of the algorithm at a time and adding an invariant to verify if everything continues to work as expected. After creating all the protocol blocks through this refinement process, we add the actual algorithm's properties. When the model checker finishes successfully, we begin to insert bugs and verify that the model checker catches the violation. Through this process we verified that all of ANTUNES' algorithms had problems, some of which were subtle, and only displayed under certain circumstances. Next, we discuss some important points in the process.

Taming the beast

The specification process helps to understand the problem in-depth, helps build small models, and is easier to write and debug than a distributed system. Such a tool provides us with a playground to test and iterate ideas more quickly, where we have a complete description of why each failure happens. Testing ideas through implementation might be neither fast nor concise on failure details. Summing up all these possibilities gives confidence in the algorithm's (and design) correctness.

Space and Time

Although these tools really help, they are limited when executing larger models (ONGARO, 2014). A specification can be too difficult to verify because of the number of states it generates and the amount of storage necessary, making larger models impractical to check without dedicated infrastructure. But such limitations are not an excuse for not using them at all. Some problems we found in ANTUNES' algorithms were subtle, whereas finding them without these tools would be an immense effort. In this regard, APALACHE, a TLC alternative, might be a solution to tackle this limitation. Apalache applies a symbolic evaluation, not explicitly enumerating all states like TLC (KONNOV; KUKOVEC; TRAN, 2019).

Types

Type errors are complex to handle during development. The structures can be sets, sequences, numbers, and others, but there is no type assertion built-in, so a variable that starts as a set could then change to an integer. The model checking will eventually fail because of an invalid transformation, but the error message may not be so explicit about the problem. We found that it is common to add a type-check invariant to circumvent this problem. APALACHE uses type annotations to infer the types (KONNOV; KUKOVEC; TRAN, 2019).

Tooling

We set up code editors to help during specification writing. These editors embed TLC and can run the model checker directly for the editor, quickly checking a specification, and providing code completion and syntax highlighting. Besides, one of the best features is that these editors handle TLC's output for violated invariants, showing the complete steps more understandably. The violation timeline helps to see the state changing and what is failing.

The resulting TLA$^+$ specifications are available in Appendix A and, while they did serve their purpose, they can definitely be improved, for example to model incorrect processes.

4.4.1 Time Flies

We had a lengthy and convoluted discussion about TLA and the temporal operators in Chapter 2.4. Now that we have the corrected algorithms, we know how they work and their properties, then we can start connecting with what we saw earlier. Then we will further the discussion on the properties of the algorithms.

Here we can discuss how the algorithm's properties relate to the TLA properties. We may include some TLA$^+$ snippets to ease the discussion with a visual aid, where the complete specifications are available in Appendix A. We start with the algorithm properties, connecting them to the operators and the system's properties. Then we discuss the fairness in our specification and why it is needed.

Liveness

The Generic Multicast algorithm has the liveness properties, namely *Validity* and *Agreement*, presented in Chapter 2 on page 30. These properties assert that the algorithm progress and that something good eventually happens (LAMPORT, 1994b). Without these properties, the system could hang forever, doing nothing, so these properties ensure that the algorithm delivers messages at some point. Now we write the properties in plain English and our TLA$^+$ specification of each one, showing the temporal operators in use. For simplicity, we display only the properties for the Generic Multicast 0 here, whereas the properties for the other algorithms are available in the appendixes.

The TLA$^+$ snippets use variables holding global information derived from the initial constants. The set *AllMessages* contains all messages in the system, whether sent or unsent; the *SentMessages* set has all messages sent in the algorithm, where $SentMessages \subseteq$

AllMessages; and the set *CorrectProcesses* with all processes that are correct in the system. The *WasDelivered* is a boolean-valued expression that checks if the given process delivered the message. We do this by checking the process' *Delivered* set.

Algorithm 4.13 shows the TLA$^+$ for the Validity property. We use the $\leadsto$ operator, asserting that, for all messages sent, if the originator is a correct process, eventually, there is a process in the message's destination that delivers the message. We can rewrite this formula using the $\Box$ and $\Diamond$ operator as $\Box(F \implies \Diamond G)$ (LAMPORT, 1994b).

Algorithm 4.13 Validity property in TLA$^+$.

$$
\begin{array}{l}
\mathit{Validity} \triangleq \\
\quad \forall\, m \in \mathit{AllMessages}: \\
\qquad m.o \in \mathit{CorrectProcesses} \leadsto \exists\, q \in m.d : \mathit{WasDelivered}(q,\ m)
\end{array}
$$

Algorithm 4.14 shows the TLA$^+$ for the Agreement property. We also use the $\leadsto$ operator. We assert that, for any arbitrary message, once a process delivers it, all the correct processes in the destination must eventually do, too.

Algorithm 4.14 Agreement property in TLA$^+$.

$$
\begin{array}{l}
\mathit{Agreement} \triangleq \\
\quad \forall\, m \in \mathit{AllMessages}: \\
\qquad \forall\, p \in \mathit{Processes}: \\
\qquad\quad \mathit{WasDelivered}(p,\ m) \leadsto \forall\, q \in \{x \in m.d : x \in \mathit{Processes}\} : \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{WasDelivered}(q,\ m)
\end{array}
$$

We finish our summary of the algorithm's liveness properties. We tested each property isolated from one another, selecting a single one to execute at a time by TLC as a system property. Some sets are static throughout the complete test, for example, the set *CorrectProcesses*. Dynamic sets, when fit, could be a better approach, for example, processes crashing, but this could create an enormous state space.

Safety

The *Partial Order*, *Collision*, and *Integrity* properties are safety properties. These properties are valid without any fairness assumptions (OWICKI; LAMPORT, 1982).

Algorithm 4.15 is the TLA$^+$ implementation for the Integrity property. We assert that, for all system messages and processes, once the process delivers a message, it did it only once and was in the destination of a sent message. The predicate *DeliveredOnlyOnce* filter the process' *Delivered* set, which holds tuples in the form of $\langle \mathit{Index}, \mathit{Message} \rangle$, and only a single message exists.

Since the Partial Order and Collision properties need to know the message's delivery instant, we created a predicate called *DeliveredInstant*, which has the process and message as arguments. Algorithm 4.16 is the TLA$^+$ representation of the Partial Order property.

Algorithm 4.15 Integrity property in TLA$^+$.

$$
\begin{array}{l}
\mathit{Integrity} \triangleq \\
\quad \Box \forall\, m \in \mathit{AllMessages}: \\
\qquad \forall\, p \in \mathit{Processes}: \\
\qquad\quad \mathit{WasDelivered}(p,\, m) \implies (\mathit{DeliveredOnlyOnce}(p,\, m) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land p \in m.d \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land m \in \mathit{SentMessages})
\end{array}
$$

Instead of writing a single big formula, we split the expression between the left-hand and right-hand sides, referenced as *LHS* and *RHS*, where *LHS* implies *RHS*. *LHS* verifies that the processes are in the messages' destinations, the message pair do not commute, and the processes deliver both messages. *RHS* checks that the message delivery order for both processes is the same. We assert that this is always valid for all processes and messages.

Algorithm 4.16 Partial Order property in TLA$^+$.

$$
\begin{array}{l}
\textsc{local}\ \mathit{BothDelivered}(p,\, q,\, m,\, n) \triangleq \\
\quad \land\ \mathit{WasDelivered}(p,\, m) \land \mathit{WasDelivered}(p,\, n) \\
\quad \land\ \mathit{WasDelivered}(q,\, m) \land \mathit{WasDelivered}(q,\, n) \\
\\
\textsc{local}\ \mathit{LHS}(p,\, q,\, m,\, n) \triangleq \\
\quad \land\ \{p,\, q\} \subseteq (m.d \cap n.d) \\
\quad \land\ \mathit{CONFLICTR}(m,\, n) \\
\quad \land\ \mathit{BothDelivered}(p,\, q,\, m,\, n) \\
\\
\textsc{local}\ \mathit{RHS}(p,\, q,\, m,\, n) \triangleq \\
\quad \land\ \textsc{let} \\
\qquad pm \triangleq \mathit{DeliveredInstant}(p,\, m) \\
\qquad pn \triangleq \mathit{DeliveredInstant}(p,\, n) \\
\qquad qm \triangleq \mathit{DeliveredInstant}(q,\, m) \\
\qquad qn \triangleq \mathit{DeliveredInstant}(q,\, n) \\
\quad\ \textsc{in} \\
\qquad \land\ (pm < pn) \equiv (qm < qn) \\
\\
\mathit{PartialOrder} \triangleq \\
\quad \Box \forall\, p,\, q \in \mathit{Processes}: \\
\qquad \forall\, m,\, n \in \mathit{AllMessages}: \\
\qquad\quad \mathit{LHS}(p,\, q,\, m,\, n) \implies \mathit{RHS}(p,\, q,\, m,\, n)
\end{array}
$$

Algorithm 4.17 is the TLA$^+$ representation of the newly added Collision property. We check that, for all processes, if it is in the destination of a pair of already delivered non-commuting messages, then the instant of each delivery is different.

All these properties use the $\Box$ operator because they must always be valid. The algorithm violates the property if the formula evaluates to false for any reason whatsoever.

Algorithm 4.17 Collision property in TLA$^+$.

$$
\begin{array}{l}
\mathit{Collision} \triangleq \\
\quad \Box \forall\, p \in \mathit{CorrectProcesses}: \\
\quad\quad \forall\, m,\ n \in \mathit{AllMessages}: \land\ m.id \neq n.id \\
\quad\quad\quad \land\ p \in (m.d \cap n.d) \\
\quad\quad\quad \land\ \mathit{WasDelivered}(p,\ m) \\
\quad\quad\quad \land\ \mathit{WasDelivered}(p,\ n) \\
\quad\quad\quad \land\ \mathit{CONFLICTR}(m,\ n) \implies \mathit{DeliveredInstant}(p,\ m) \neq \mathit{DeliveredInstant}(p,\ n)
\end{array}
$$

In our tests, the sets were static at all times. These properties could take advantage of simulating incorrect processes, asserting the algorithm's fault tolerance.

Fairness

We presented fairness in Chapter 2 in Section 2.4.1 on page 34. Informally, for any of the specification's steps, it either stutters or the state changes. In our work, all specifications rely on the system's weak fairness and may not work without it.

Algorithm 4.18 shows all our specifications entry point in TLA$^+$. This predicate has the *Init* to initialize the structures and an action that accepts stuttering steps on the *vars* state. Line 2 extends the predicate by adding the liveness requirements. Without the weak fairness, our algorithm could stutter forever and never deliver a message, a violation of our liveness properties, Validity, and Integrity.

Algorithm 4.18 Specification `Spec` definition.

$$
\begin{array}{l}
\mathit{Spec} \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \\
\qquad\qquad \land\ \mathrm{WF}_{\mathit{vars}}(\exists\, \mathit{self} \in \mathit{Processes} : \mathit{Step}(\mathit{self}))
\end{array}
$$

4.4.2 Withal Thine Basic Properties

The work of PEDONE; SCHIPER presents an algorithm to solve the Generic Broadcast problem and discusses two properties, *deliver latency* and *strictness.* We now bring these two properties to our proposals. Here we introduce the propositions leaving the proofs for future works.

4.4.2.1 Delivery Latency

Introduced to measure the efficiency of algorithms solving a Broadcast problem (PEDONE; SCHIPER, 1999), we will use it in our algorithms. Informally, delivery latency is the number of events a message m goes through from sending to delivery in a run R of an algorithm A solving the Multicast problem, written as $dl^R(m)$ (PEDONE; SCHIPER, 1999). The delivery latency bases itself on a modified Lamport's clock (LAMPORT, 1994b), where we have (PEDONE; SCHIPER, 1999):

- A *send* and a *local* event on process p do not modify p's clock.
- $ts(send(m))$ is the timestamp of a $send(m)$ event, and $ts(m)$ the timestamp carried by message m, such as $ts(m) = ts(send(m)) + 1$.
- The timestamp of $receive(m)$ on process p is the maximum between $ts(m)$ and p's clock value.

Let $\mathcal{L}_r\langle m\rangle$ be the set of all processes that, for message m, $\boxed{\text{gm-Delivered } m}$ in run R of algorithm A, and $\mathcal{D}_p\langle m\rangle$ the $\boxed{\text{gm-Delivered } m}$ event at process p. The definition of delivery latency of m in run R is in Equation 9a.

$$dl^R(m) \triangleq \max(\{ts(\mathcal{D}_p\langle m\rangle) - ts(\text{gm-Send}\langle m\rangle) : p \in \mathcal{L}_r\langle m\rangle\}) \tag{9a}$$

We now define the propositions for our algorithms. We follow the same approach as PEDONE; SCHIPER for simplicity, using runs of a single message. We assume that there is an implementation for the Atomic/Generic Broadcast and process communication available and that the delivery latency of these algorithms is 1. Proposition 1 defines a lower bound for the delivery latency of our algorithms. Proposition 2 proposes that groups with a synchronized clock reach this lower bound.

Proposition 1. *Atomic/Generic Broadcast is a primitive available for the algorithm. If R_A is a set of runs generated by an algorithm A that solves the Generic Multicast problem such that only a single message $m \in \mathcal{M}$ is $\boxed{\textit{gm-Send } m \textit{ to } \mathcal{G}}$ and $\boxed{\textit{gm-Delivered } \_}$ and $|\mathcal{G}| > 1$, then there is no run R in R_A where $dl^R(m) < 2$.*

Proposition 2. *Atomic/Generic Broadcast is a primitive available for the algorithm. If R_A is a set of runs generated by an algorithm A that solves the Generic Multicast problem such that only a single message $m \in \mathcal{M}$ is $\boxed{\textit{gm-Send } m \textit{ to } \mathcal{G}}$ and $\boxed{\textit{gm-Delivered } \_}$, $|\mathcal{G}| > 1$, and groups in m's destination have the same ts(ab-Delivered $\langle m\rangle$), then there is a run R in R_A where $dl^R(m) = 2$.*

Our intuition for Proposition 1 is that, when sending a message to more than one group, there is the initial Atomic/Generic Broadcast and then the proposals exchange, meaning it is impossible to have a delivery latency of less than two. And for Proposition 2, if the addressed groups are tightly synchronized, then only needed the first Atomic/Generic Broadcast and the proposals exchange, avoiding the broadcast for synchronization. Observe that none of this does apply when addressing only a single group because it skips the proposals exchange and the synchronization broadcast.

Currently, our algorithm delivers all messages with the same delivery latency. The initial goal was to keep the processes' clock as low as possible (ANTUNES, 2019), whereas ours was to formalize and correct the algorithm. Future work could focus on introducing optimizations. For example, could we avoid the second Atomic/Generic Broadcast to synchronize the local group for non-conflicting messages?

4.4.2.2 Strictness

The Generic Multicast problem can be solved using an Atomic Multicast implementation, but with unnecessary ordering in messages. If the message ordering adds a cost, we should work to keep the cost as low as possible (PEDONE; SCHIPER, 1999). An algorithm that solves Atomic Multicast can have a *Strictness* property, identifying that it avoids unnecessary message ordering. We do not enforce this property because spontaneous orders might happen. We define Strictness as (PEDONE; SCHIPER, 1999):

- **Strictness**: Algorithm $A_{\mathcal{C}}$ is an algorithm that solves Generic Multicast problem with the conflict relation $\mathcal{C} \subset \mathcal{M} \times \mathcal{M}$, and R_A is the set of runs of $A_{\mathcal{C}}$. There exists a run R in R_A where messages $m_1, m_2 \in \mathcal{M}$ and $m_1 \nsim m_2$, and processes in Π gm-Delivered m_1 and m_2 in a different order.

We use TLA$^+$ to verify this property, using proof by contradiction. We write a property to check that, using a conflict relation $\mathcal{C} \subset \mathcal{M} \times \mathcal{M}$, all processes deliver the message in the same order. TLC provides a counter-example of a violation, that is, there exists a run $A_{\mathcal{C}}$ where our algorithm delivers messages in a different order.

4.4.2.3 Genuineness

The minimality property defined in Section 2 ensures that only necessary processes participate in the message delivery, that is, the sender and destination. Although the property is for an algorithm that solves the Atomic Multicast problem, we can also extend this to the algorithms that solve the Generic Multicast problem. All the algorithms we presented here are genuine, meaning all algorithms provide the minimality property. Proposition 3 states this for our algorithms, where we left the proof for future works.

Proposition 3. *Generic Multicast 0, 1, and 2 are genuine algorithms that solve the Generic Multicast problem.*

4.4.2.4 Quiescence

Another property is for an algorithm to be quiescent. A quiescent algorithm is an algorithm that eventually stops sending messages (AGUILERA; CHEN; TOUEG, 2000). Using failure detectors, algorithms that only tolerate process crashes can become quiescent and tolerate both process crashes and message losses (AGUILERA; CHEN; TOUEG, 2000), so the algorithms we present here can be made quiescent.

We define Proposition 4, Proposition 5, and Proposition 6 state that if the communication primitives our algorithms are using are quiescent, then our algorithms are quiescent.

Proposition 4. *Generic Multicast 0 is a quiescent algorithm that solves the Generic Multicast problem if the process communication is quiescent.*

Proposition 5. *Generic Multicast 1 is a quiescent algorithm that solves the Generic Multicast problem if the process communication and Atomic Broadcast are quiescent.*

Proposition 6. *Generic Multicast 2 is a quiescent algorithm that solves the Generic Multicast problem if the process communication and Generic Broadcast are quiescent.*

Our intuition for all these propositions comes from the fact that our algorithm does not have any mechanism that infinitely sends messages. We use the Atomic/Generic Broadcast for group communication and the channel that connects the processes, so once no more execution of $\boxed{\text{gm-Send } m \text{ to } \mathcal{G}}$ happens, the algorithm stops sending messages. Therefore, if the underlying primitives are quiescent, our algorithms are too.

Chapter 5

Generic Multicast Implementation

This chapter discusses the prototype implementation of the Generic Multicast 1 using Golang. We chose this algorithm because it is a better study case since Generic Multicast 0 only introduces the generalized concepts and Generic Multicast 2 would require more work implementing the Generic Broadcast.

Even though the current implementation is not production-ready, the design and implementation of a consensus algorithm can be a non-trivial yet engaging task. Gaps can exist between the protocol definition to what it would be in a real-world production environment. These gaps could lead to engineers implementing a protocol that differs from the specification, leading to an implementation that is not verified to be correct (CHANDRA; GRIESEMER; REDSTONE, 2007).

We start this chapter with a high-level overview of the architecture, how components interact, and the requirements beyond the ones needed by the algorithm. Section 5.2 describes the base communication primitives and the message's format. Section 5.3 describes the algorithm core implementation and the converted data structures from the specification to code. Section 5.4 discusses the tests and how we verify the prototype. Then Section 5.5 concludes the chapter with a summary and future improvements.

5.1 The Bricks in the Foundation

This section discusses the architecture at a high-level. The complete architecture is in Figure 4, serving as a reference throughout the architecture explanation, organized in a layered architecture. We describe the components, their interactions, and which ones are required. The most north is the actual application that wants to replicate any information. The middle is the Client Level layer, exposing an API for the application to interact with the algorithm. The bottom layer is the Protocol Level, the actual protocol implementation, along with other components necessary to work. We added reference points to show the components' interactions. The interactions include method invocation, communication through Golang's channels, and network interactions.

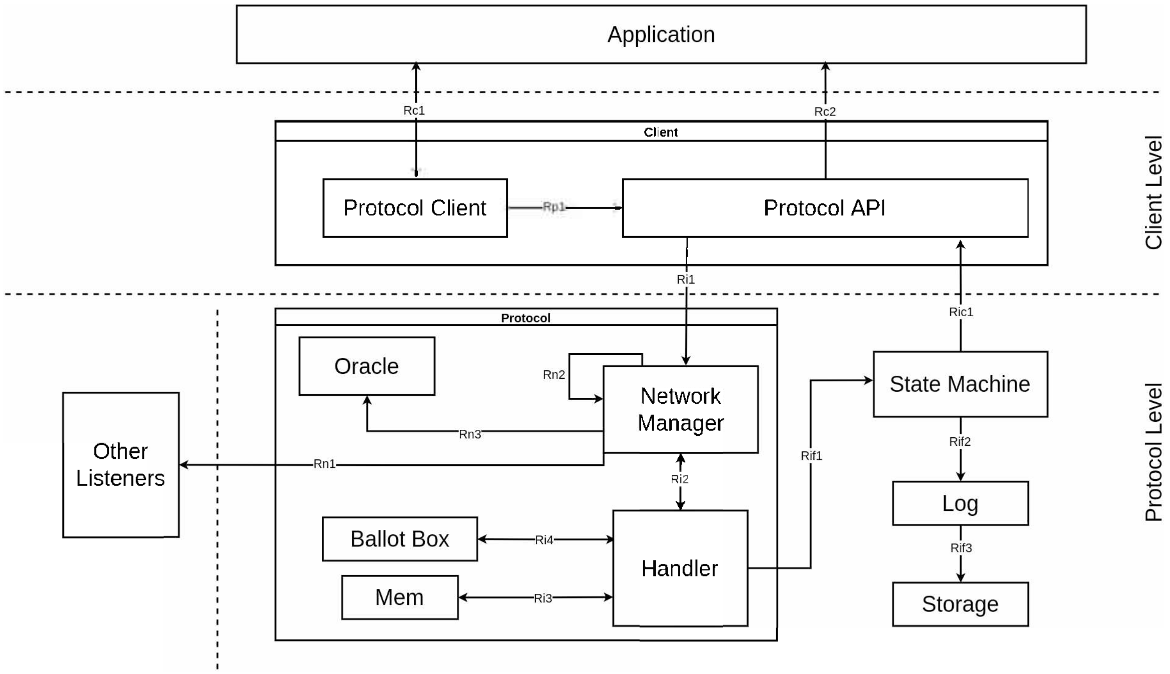

Figure 4 – Implementation architecture.

Starting at the Client Level, there are two components, the *Protocol Client* and the *Protocol API*. The Protocol Client through the `Rc1` reference point is the only way the application can interact with the protocol. The application can subscribe to a channel to receive notifications about data replication, issue requests to the protocol, and terminate the client. All of these interactions are through the `Rc1` reference point. The Protocol API is the one that interacts directly with the Protocol Layer. Reference point `Ri1` sends an asynchronous message to the protocol, and reference point `Rc2` send notifications to a subscribed application.

The Protocol Client, Protocol API, and references compose the Client Level. Interactions between the application and the protocol will always pass through this level. This layer is a user-facing interface, not an algorithm requirement, but this helps when developing integration tests.

5.2 Do They Talk?

The current algorithm requires two primitives, communication between processes and Atomic Broadcast. In this section, we describe how we implemented these primitives. The primitives an algorithm requires are a crucial component and must provide all the guarantees.

In Figure 4, the Network Manager encapsulates the primitives. It receives and sends messages from both primitives. The manager creates a socket and starts a goroutine to consume incoming messages. Messages are processed asynchronously, except when received through Atomic Broadcast.

5.2.1 The Process Talk

We start with communication between processes. The primitive is in a dedicated open-source project[1]. We implemented a TCP server to send and receive messages using the Go networking package (GOLANG, 2021a). The server is a concrete implementation of an interface from the Go package.

The implementation itself does not try to provide anything too complicated. We implement what is needed, avoiding details like retries, buffering to reduce syscalls, and fancy serialization. The implementation contains basic configuration properties, like server port and address, asynchronous actions' timeout, and pool size.

For a process to send a message using the primitive, it must know the destination's address. Sending a message can be complicated since a message can have any destination, so each process must know all other processes' addresses. To solve this problem, we use the *Oracle* component, which converts a group alias and returns a collection of all addresses within that group. The application provides a concrete Oracle instance. The messages we exchange in the protocol reference only groups and use the Oracle to translate to addresses.

5.2.2 In Totally Ordering

We built the Atomic Broadcast primitive as a separate open-source project[2] on top of etcd[3]. The primitive implementation is also very straightforward but has some points of attention that could harm the correct behavior. First, we describe etcd and then our implementation.

The etcd is a strongly consistent, distributed key-value store (ETCD, 2021) that uses Atomic Broadcast, implementing the Raft protocol (ONGARO; OUSTERHOUT, 2014) that provides strong consistency. This implementation is well established and used in multiple production environments and open-source projects (ETCD, 2021). A client interacts with etcd by connecting to a server and issuing remote procedure calls (ETCD, 2021). Multiple APIs are available, but we use only the KV and Watch. The Jepsen test verified that the APIs we use holds the algorithm guarantees (KINGSBURY, 2020). Such a test, however, checks the presence of bugs but does not ensure their absence and the algorithm's correctness.

Through KV API is possible to manipulate key-value pairs stored in etcd (ETCD, 2021). Specifically, we use the `Put` procedure to write values associated with a key,

[1] https://github.com/digital-comrades/proletariat
[2] https://github.com/jabolina/relt
[3] https://github.com/etcd-io/etcd

causing the key's revision to increment and generating one event in the event history (ETCD, 2021). Revision is a 64-bit, cluster-wide counter that serves as a global logical clock, sequentially ordering all updates to the store and incrementing each time a key is modified (ETCD, 2021). Write operations issued to the etcd server are strict-serializable, even during pauses, crashes, clock skew, network partitions, and membership changes (KINGSBURY, 2020).

The other API used in our implementation is the Watch API, which receives notifications about changes to a single key (ETCD, 2021). Using this API, starting from a given revision number, all clients receive the same sequence of updates in the same order (KINGSBURY, 2020).

Our Atomic Broadcast implementation is an etcd client. The client configuration includes information about the etcd server to connect to and the group to which it belongs. So, how is all this put together? Communication happens by listening to a key for changes and writing values associated with a key. Broadcasting a message to a group means writing the message object using the group's name as a key. To receive messages, the processes within the group use the Watch API to listen to the key with the group name. The notifications are consistent and have a total order (KINGSBURY, 2020).

One of the configurable values is the timeout for some operations. The client has a maximum time frame to consume messages. The consumption time-bound and the fact that we do not implement retries could lead to message loss when a timeout happens. This problem can have a complex fix, but since our implementation is only a prototype to run in a controlled environment, timeouts did not actually occur. We did not verify what happens when a timeout occurs, but the most likely outcome is a violation of the algorithm properties.

The development experience was not overly-complicated, the etcd documentation is complete (ETCD, 2021), and examples are easy to find. The implementation to interact with the etcd server was only a few lines. Most of the development effort was to build the structure around the etcd client, managing goroutines, configurations, and a simple user API. The only more complicated problems were due to the gRPC (GOOGLE, 2021) dependency conflicting with the one used by etcd.

5.2.3 Messages

Beyond the transport definition, there are also the requirements for the message itself. Since transport is agnostic to what it is transporting, the message format can be arbitrary but must meet the protocol's needs. The protocol requires that the messages have a strict total order, which the protocol uses to break ties between timestamps.

We embed a 128 bits random identifier into messages. We allocate 128 bits and convert them into a string. We rely on the probability of selecting duplicated 128 bits being negligible to avoid collision between identifiers. Using a tool dedicated to generating identifiers would be a better approach.

Listing 5.1 displays the complete message object. We can see that we transport redundant information. Future work could create a proper format carrying minimal information while keeping semantics. There is also room to improve the serialization, where we currently use the default available in Golang.

Listing 5.1 – Message format definition.

```
Message {
    identifier: String
    header {
        version: Int
        type: ABSend | Send
    }
    content {
        meta {
            timestamp: UInt64
            identifier: String
        }
        operation: command | query
        content: Array[Byte]
        extensions: Array[Byte]
    }
    state: 0 | 1 | 2 | 3
    timestamp: UInt64
    destination: Array[String]
    from: String
}
```

5.3 At the Core

Now we discuss the implementation, where we implement the version that does not use the Reliable Multicast primitive. We rely on the components introduced in the previous sections. The complete algorithm implementation includes the *Protocol* container in Figure 4 on page 76.

The Network Manager receives a message, passes it through `Ri2` to be processed, and executes the procedure's callback. The message's header identifies from which primitive it arrived, so the algorithm knows the corresponding step. The return of each procedure is an enumeration that points to the next step, for example, sending the updated message using the Atomic Broadcast primitive to the current group. We use this approach to detach the networking use from the core algorithm. The insertion in the *Mem* structure also executes after the message processing.

One last detail is the implicit data structure that holds the timestamp proposals for each message. This structure is seen in Algorithm Algorithm 4.8 on line 20 on page 60 when choosing the maximum value. We call the structure *Ballot Box*, as seen in Figure 4 on page 76. The *Ballot Box* keeps the proposals until it has a vote from each group in the message's destination, and once the final timestamp is selected, it discards the proposals. Notice that our implementation does not discard delayed votes. It is possible to receive timestamp proposals even after delivering the message since the channels have an arbitrary delay.

5.3.1 In Mem Store

We describe our *Mem* structure implementation in this section. During the algorithm execution, the *Mem* structure stores the messages currently being processed and participates in all procedures, where its performance is crucial to how well the algorithm operates. Before starting development, we defined some requirements for the implementation:

- No duplicated entries, where duplications could lead to liveness problems or delivering messages more than once;
- Thread-safe, it should handle concurrent requests;
- Low time complexity, the operations should not take too much time.

Our implementation is modular, breaking the problems into smaller ones to solve in each module. Figure 5 shows the structure modules and the organization. All the interactions with the *Mem* structure are through an interface with the same name. We store messages currently processing in the *Processing* region, and the finished on *Processed*. The components work together to achieve the goals above.

The memory paradise

The first component is the *Processing* region for messages currently being processed. The underlying store structure is a priority queue, holding the smallest element in the head, sorting by the timestamp and the unique identifier when needed.

We implement the priority queue using a Fibonacci Heap. The most common instructions, `findMin`, `insert`, and `decreaseKey`, have a time complexity of $O(1)$, and for `delete`, it is $O(\log n)$. There is an additional instruction for scanning the structure for non-conflicting messages on state `S3` to be delivered, which has the complexity

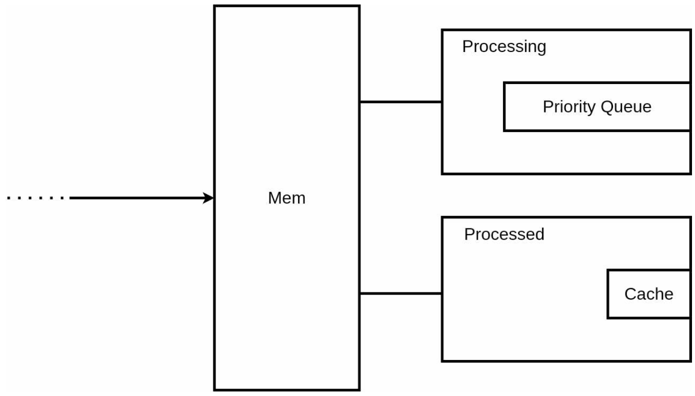

Figure 5 Implementation architecture of the *Mem* structure.

$O(n^2)$, where each message checks for conflict against all others. We execute the scan asynchronously when updating a message with state S3 or during the delivery execution, although the operation must acquire a read lock. Note that besides the structure being a queue, it is possible to remove elements from any arbitrary position.

The queue works reactively, taking the initiative to notify about messages ready for delivery, avoiding the need for constant verification. Since the structure is a priority queue, we only check the head element, and we do so after every change in it. We ensure thread safety by using a read-write lock.

The purged ones

The other region, called *Processed*, serves a simple but necessary purpose, to avoid duplicating notifications about messages ready for delivery. Instead of *Processing* to keep track of notified elements, the *Processed* handles this. After issuing a notification, we insert the message's unique identifier into the cache with a time-to-live of 10 minutes. The time of 10 minutes is an arbitrary value, and possibly, we could reduce this value to only a few seconds without affecting the algorithm. Another solution is to remove the element after removing the message from the queue.

With all these modules behind the *Mem* structure, we met all requirements. But still, there is room for improvement. Reduce the memory footprint on the stored elements. Future work could reduce the object, keeping only necessary information instead of the complete message. Try to apply a lock-stripping on the *Processing* region to reduce lock contention, but this could easily lead to a complex implementation.

5.4 Thy Elden Tests

To verify the algorithm implementation, we developed multiple tests. We use unit tests to check components in isolation, but our focus here is on the integration tests. The

Go runtime provides a package for developing automated tests and a data-race detector. Although the language's concurrency primitives aid in writing concurrent code, sometimes they are not enough (GOLANG, 2022b). The data-race detector identifies such conditions only if they trigger during execution, so load and integration tests are a valuable place to enable it (GOLANG, 2022b). Observe that these tests guarantee neither the algorithm's correctness nor the absence of bugs.

The tests execute in isolation but share the same host and resources. We create a testing harness, which we refer to as *Unity*, which has three groups of three processes each. Each action the test executes selects one of the groups available, then one of the processes in the group, and then effectively applies the action. The Unity chooses groups and processes in a round-robin approach.

The Unity sends a message by selecting an initiator and issuing an asynchronous request. To verify the delivery order, Unity pins one process, retrieving its messages, and compares the sequence against the other groups' sequences. We compare only against a single participant in each group.

We execute the tests for every new code added. The environment is based on Linux, requiring a Go installation and an etcd server running. Network usage relies on the loopback network, never executing requests outside the machine that runs the tests. Tests use an in-memory store only.

5.4.1 The Elder Logs

To help during tests, we implement a write-ahead log (WAL) structure to hold the delivered messages. A WAL is an ordered sequence of commands, adding new received commands to the end. This implementation leads to additional components, all displayed in Figure 4 on page 76 as the components right of the Protocol container, including *State Machine*, *Log*, and *Storage*. We query the Log structure during the tests to verify the ordering between groups.

Take Figure 4 on page 76 as a reference guide. After the algorithm delivers a message, it will synchronously invoke the State Machine to commit a new entry through `Rif1`. The State Machine calls the Log through `Rif2` to add the message to the WAL, and once complete, the State Machine notifies the user through `Ric1`. The State Machine is responsible for handling the Log and notifying about committed entries.

5.4.2 Taking the Test

We develop the integration tests to verify if the algorithm's properties also hold for the implementation. All tests follow a similar approach, sending contrived messages and then using the Log structure to check the delivery order. The check varies with which of the algorithm's properties we are testing.

The Agreement and Integrity properties share the same test. The test broadcasts a single message, and after confirming the delivery, verifies that all members' WAL contains the same single message. The test for the Validity property issues a single request and waits for the commit notification from the State Machine, whereas not necessary to compare the delivered value among members.

The Partial Order property has a more elaborate test. The test suite continues using three groups, now referenced as `A`, `B`, and `C`. The test sends a collection of messages varying the destination, `AB`, `BC`, `AC`, and `ABC`. Then we need to verify the delivery order. For example, groups `A` and `B` must have the same order for messages sent to `AB` and `ABC`, validating all intersecting groups. We do not test the Collision property since the WAL already makes messages to have an order.

The remaining tests check variations in the message's destinations and conflicts. That is, we test the broadcast, multicast, and generic behavior. These tests are pretty straightforward and hold some similarities with the property ones. Broadcast check order on all groups, multicast in intersecting groups, and the generalized on conflicting messages.

The current integration test suite covers the properties and some of the behaviors. Future work could focus on creating Jepsen tests. Such tests would increase confidence in the algorithm implementation when encountering different hazards.

5.5 Journey So Far

We covered all details about the prototype implementation. We can now conclude by discussing the experience of implementing an algorithm directly from a specification and summarizing improvements for future works.

We started the implementation directly from ANTUNES' proposal without verifying it in TLA$^+$. Our goal was to implement something that did not violate the Generic Multicast properties instead of blindly following the algorithm. Not before long, our prototype was different from the original proposition.

The implementation without a proper specification was complex, even if for just a prototype. Some requirements were already defined beforehand, for example, the communication primitives, so we had plenty of solid ground to begin before even starting with the algorithm. After we started with the algorithm, we entered into a process of executing the tests, debugging, and guessing what may be causing the failures. Iterate this process without a clue as to why some changes were necessary, and soon we start to patch holes instead of fixing the root cause.

Once we finished the first version of the prototype, we had some leads on problems with the original algorithm, so then we turned our attention to writing a TLA$^+$ specification. For example, we were aware that messages could go back in time when we started with TLA$^+$, but we did not know the cause yet. With just a few experiments, we identified that we could solve the problem by verifying if the message exists in *Mem* before executing the method to select a message's final timestamp.

With the problems uncovered in TLA$^+$, we went back to fixing the prototype. Following the specification was a much better experience in the implementation. There are still some difficulties in implementing something directly from the specification, for example, correctly implementing some abstract data structures or handling concurrency properly. Details like these are for implementors to decide, but just detailing how a data structure should behave would greatly help the implementor's decisions.

Implementing an algorithm and writing a TLA$^+$ specification, simultaneously or not, help understand the underlying problem, identify core properties, design proper data structures, and decision making. We had a good experience using the Go language. Features for concurrency and testing shipped with the language and a large ecosystem with libraries for distributed systems helped us get started quickly. The only problem, if we can say so, was regarding the transitive dependency when using etcd, but other than that, we did not have any issue around the tools and could focus only on developing the prototype.

All That Glitters

Besides the enjoyable development experience, there are improvements left for future work. We use this section to summarize everything. These improvements focus on making the implementation more production-like, and some could be complex to implement.

We begin with the communication between processes. The improvements include adding retries when failing to send messages with a configurable retry policy. Apply some techniques to reduce system call. Lastly, improve the serialization of the message.

Now for the Atomic Broadcast primitive. The client consuming a message being a time-bound operation can lead to message loss, which affects the correctness of the primitive. We could try to use etcd's transactions to tackle this, keeping track of the revision number of the last item consumed by the client.

The remaining improvements now apply to the algorithm implementation. Most of these refer to the message object, reducing the size to transfer over the network and stored in the *Mem* structure; better generation for the unique identifier; and improved serialization. For the core algorithm implementation, fix the procedures' atomicity. Improvements for the *Mem* structure include reducing lock contention and re-arranging the design to remove the need for a *Processed* region. The tests also can take advantage of some improvements, expanding the suite with more scenarios and failures simulation.

Chapter 6

Conclusion

Distributed systems algorithms' correctness is crucial. The current work verifies three algorithms proposed by ANTUNES using TLA$^+$. We found subtle problems in each one, which makes it clear that only reasoning may not be enough for some algorithms, that apart from closely resembling the source algorithms, issues can still exist. A more robust verification may be necessary, which only helps in increasing the confidence in the algorithm's correctness.

We take a step in describing how we applied TLA$^+$ in the verification, where all specifications are openly available[1]. We verified all of ANTUNES' algorithms and corrected all problems encountered, and at this stage, we did not try to introduce optimizations. The most noteworthy change was the removal of the Reliable Multicast primitive for Generic Multicast 1 and Generic Multicast 2. We also explore additional properties the algorithms have, which are: Strictness, Minimality, and Quiescence.

Lastly, we implemented a prototype of Generic Multicast 1[2]. Even for a prototype with a study purpose, it was a challenge. Implementing and specifying the algorithms was an enlightening process that helped us to deeply understand how the algorithms work and how the properties fit together. Starting from a specified algorithm could reduce the development time since the developer can focus on the programming problems without worrying about the algorithm's correctness.

The current work has limitations on the specifications and the implemented prototype. Our specification generates too many states for larger models, making some scenarios impractical for verification. We could devote some effort to using another type of model checker so that verifying larger models is possible. We defined propositions for some additional properties, whereas writing proof for these is left for future work.

The algorithms can serve as a base to create algorithms that fit a real-world production environment. Such a change would start with adapting the algorithm to work with dynamic groups. This change would allow for processes to join and leave as they please.

1 <https://github.com/jabolina/mcast-tlaplus>
2 <https://github.com/jabolina/go-mcast>

Introducing and formalizing optimizations is a welcome contribution to the algorithms. The prototype also has room for improvement, strengthening the implementation so other applications can use it as a foundation to create more robust algorithms.

Bibliography

AGUILERA, M. K.; CHEN, W.; TOUEG, S. On quiescent reliable communication. **SIAM Journal on Computing**, SIAM, v. 29, n. 6, p. 2040–2073, 2000. <https://doi.org/10.1137/S0097539798341296>.

AHMED-NACER, T.; SUTRA, P.; CONAN, D. The convoy effect in atomic multicast. In: IEEE. **2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)**. [S.l.], 2016. p. 67–72. <https://doi.org/10.1109/SRDSW.2016.22>.

ALPERN, B.; SCHNEIDER, F. B. Recognizing safety and liveness. **Distributed computing**, Springer, v. 2, n. 3, p. 117–126, 1987. <https://doi.org/10.1007/BF01782772>.

ANTUNES, D. **Fault-Tolerant Generic Multicast Algorithms for Wide Area Networks**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2019. Internal Communication.

BEERS, R. Pre-rtl formal verification: an intel experience. In: **Proceedings of the 45th annual Design Automation Conference**. [S.l.: s.n.], 2008. p. 806–811. <https://doi.org/10.1145/1391469.1391675>.

BIRMAN, K. P.; JOSEPH, T. A. Reliable communication in the presence of failures. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 5, n. 1, p. 47–76, 1987. <https://doi.org/10.1145/7351.7478>.

BLASGEN, M. et al. The convoy phenomenon. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 13, n. 2, p. 20–25, 1979. <https://doi.org/10.1145/850657.850659>.

BORNHOLT, J. et al. Using lightweight formal methods to validate a key-value storage node in amazon s3. In: **Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles**. [S.l.: s.n.], 2021. p. 836–850. <https://doi.org/10.1145/3477132.3483540>.

BUTCHER, M.; FARINA, M. **Go in Practice**. [S.l.]: Manning Publications Company, 2016. ISBN 9781633430075.

CAMARGOS, L. **Multicoordinated agreement protocols and the log service**. Tese (Doutorado) — Università della Svizzera italiana, 04 2008. <https://doi.org/10.1109/NCA.2008.28>.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: **Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing**. [S.l.: s.n.], 2007. p. 398–407. <https://doi.org/10.1145/1281100.1281103>.

CHANDRA, T. D.; HADZILACOS, V.; TOUEG, S. The weakest failure detector for solving consensus. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 43, n. 4, p. 685–722, 1996. <https://doi.org/10.1145/234533.234549>.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 43, n. 2, p. 225–267, 1996. <https://doi.org/10.1145/226643.226647>.

CHARRON-BOST, B.; PEDONE, F.; SCHIPER, A. Replication. **LNCS**, Springer, v. 5959, p. 19–40, 2010. <https://doi.org/10.1007/978-3-642-11294-2>.

CHARRON-BOST, B.; SCHIPER, A. Uniform consensus is harder than consensus. **Journal of Algorithms**, Elsevier, v. 51, n. 1, p. 15–37, 2004. <https://doi.org/10.1016/j.jalgor.2003.11.001>.

COELHO, P. R.; SCHIPER, N.; PEDONE, F. Fast atomic multicast. In: **2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.: s.n.], 2017. p. 37–48. <https://doi.org/10.1109/DSN.2017.15>.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Totally ordered broadcast and multicast algorithms: A comprehensive survey. 2000. Disponível em: <http://infoscience.epfl.ch/record/49982>.

_____. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 36, n. 4, p. 372–421, 2004. <https://doi.org/10.1145/1041680.1041682>.

ETCD. **etcd**. [S.l.], 2021. Disponível em: <https://etcd.io/>. Acesso em: 13.07.2021.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 32, n. 2, p. 374–382, 1985. <https://doi.org/10.1145/3149.214121>.

FRITZKE, U. et al. Fault-tolerant total order multicast to asynchronous groups. In: IEEE. **Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)**. [S.l.], 1998. p. 228–234. <https://doi.org/10.1109/RELDIS.1998.740503>.

GOLANG. **Go Net**. [S.l.], 2021. Disponível em: <https://pkg.go.dev/net>. Acesso em: 13.07.2021.

_____. **The Go Programming Language Specification**. [S.l.], 2021. Disponível em: <https://golang.org/ref/spec>. Acesso em: 22.07.2021.

_____. **Effective Go**. [S.l.], 2022. Disponível em: <https://go.dev/doc/effective_go>. Acesso em: 12.03.2022.

_____. **Introducing the Go Race Detector**. [S.l.], 2022. Disponível em: <https://go.dev/blog/race-detector>. Acesso em: 12.03.2022.

GOOGLE. **gRPC**. [S.l.], 2021. Disponível em: <https://grpc.io/>. Acesso em: 14.07.2021.

GUERRAOUI, R.; SCHIPER, A. Total order multicast to multiple groups. In: IEEE. **Proceedings of 17th International Conference on Distributed Computing Systems**. [S.l.], 1997. p. 578–585. <https://doi.org/10.1109/ICDCS.1997.603426>.

KINGSBURY, K. **etcd 3.4.3**. [S.l.], 2020. Disponível em: <https://jepsen.io/analyses/etcd-3.4.3.pdf>. Acesso em: 14.07.2021.

KLEPPMANN, M. **Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems**. [S.l.]: " O'Reilly Media, Inc.", 2017. ISBN 1449373321.

KONNOV, I.; KUKOVEC, J.; TRAN, T.-H. Tla+ model checking made symbolic. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 3, n. OOPSLA, p. 1–30, 2019. <https://doi.org/10.1145/3360549>.

LAMPORT, L. **Introduction to TLA**. [S.l.]: Digital Equipment Corporation Systems Research Center [SRC], 1994.

_____. The temporal logic of actions. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 16, n. 3, p. 872–923, 1994. <https://doi.org/10.1145/177492.177726>.

_____. **Specifying systems**. [S.l.]: Addison-Wesley Boston, 2002. v. 388. ISBN 9780321143068.

_____. Generalized consensus and paxos. Citeseer, 2005. Disponível em: <https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/>.

_____. Time, clocks, and the ordering of events in a distributed system. In: **Concurrency: the Works of Leslie Lamport**. [S.l.: s.n.], 2019. p. 179–196. <https://doi.org/10.1145/3335772.3335934>.

_____. **Industrial Use of TLA+**. [S.l.], 2021. Disponível em: <https://lamport.azurewebsites.net/tla/industrial-use.html>. Acesso em: 20.05.2022.

_____. **The TLA+ Toolbox**. [S.l.], 2021. Disponível em: <https://lamport.azurewebsites.net/tla/toolbox.html>. Acesso em: 14.07.2021.

_____. **The Writings of Leslie Lamport**. [S.l.], 2022. Disponível em: <https://lamport.azurewebsites.net/pubs/pubs.html>. Acesso em: 29.08.2022.

LAMPORT, L. et al. Paxos made simple. **ACM Sigact News**, v. 32, n. 4, p. 18–25, 2001. Disponível em: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.

NEWCOMBE, C. et al. How amazon web services uses formal methods. **Communications of the ACM**, ACM New York, NY, USA, v. 58, n. 4, p. 66–73, 2015. <https://doi.org/10.1145/2699417>.

ONGARO, D. **Consensus: Bridging theory and practice**. Stanford University, 2014. Disponível em: <http://purl.stanford.edu/qr033xr6097>.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **2014 USENIX Annual Technical Conference (USENIX ATC 14)**. Philadelphia, PA: USENIX Association, 2014. p. 305–319. ISBN 978-1-931971-10-2. Disponível em: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>.

OWICKI, S.; LAMPORT, L. Proving liveness properties of concurrent programs. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 4, n. 3, p. 455–495, 1982. <https://doi.org/10.1145/357172.357178>.

PEDONE, F.; SCHIPER, A. Generic broadcast. In: SPRINGER. **International symposium on distributed computing**. [S.l.], 1999. p. 94–106. <https://doi.org/10.1007/3-540-48169-9_7>.

_____. Handling message semantics with generic broadcast protocols. **Distributed Computing**, Springer, v. 15, n. 2, p. 97–107, 2002. <https://doi.org/10.1007/s004460100061>.

REZENDE, T. F. **Uma Implementação Fiel do Algoritmo Generalized Paxos e uma CStruct para o Problema de Coordenação de Lease Distribuído**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 01 2017. <http://dx.doi.org/10.14393/ufu.di.2018.141>.

SCHIPER, A. Dynamic group communication. **Distributed Computing**, Springer, v. 18, n. 5, p. 359–374, 2006. <https://doi.org/10.1007/s00446-005-0129-4>.

SCHIPER, N.; PEDONE, F. Optimal atomic broadcast and multicast algorithms for wide area networks. In: **Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing**. [S.l.: s.n.], 2007. p. 384–385. <https://doi.org/10.1145/1281100.1281185>.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 22, n. 4, p. 299–319, 1990. <https://doi.org/10.1145/98163.98167>.

SYSTEMS, I. **Industrial Use of TLA+**. [S.l.], 2020. Disponível em: <https://github.com/informalsystems/vdd/blob/410e427533fcbc42426124effc305d02fa9786ba/guide/guide.md>. Acesso em: 24.05.2022.

TU, T. et al. Understanding real-world concurrency bugs in go. In: **Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2019. p. 865–878. <https://doi.org/10.1145/3297858.3304069>.

YU, Y.; MANOLIOS, P.; LAMPORT, L. Model checking tla+ specifications. In: SPRINGER. **Advanced Research Working Conference on Correct Hardware Design and Verification Methods**. [S.l.], 1999. p. 54–66. <https://doi.org/10.1007/3-540-48153-2_6>.

Appendix

APPENDIX A

On the Specifications

This chapter contains all the TLA$^+$ specifications. The actual TLA files are available online for public scrutiny, and the specifications here come directly from these online files. We use the TLATEX program for typesetting the TLA$^+$ modules (LAMPORT, 2002), writing the contents in this chapter as is. We will provide little to no comments about the contents since the modules are self-explanatory. We start with the communication primitives and helpers and then with the specifications. The algorithms have a slightly smaller font size to fit better on the page.

A.1 Communication Primitives

We wrote the primitives for a quasi-reliable channel for process communication, Atomic Broadcast and Generic Broadcast for group communication. To write the process communication a single time and use it in all specifications, it uses the structures as a group exists all times. The Generic Multicast 0 uses this primitive with groups with one process.

The quasi-reliable abstraction is in A.1. The Atomic Broadcast abstraction is in A.1. The Generic Broadcast abstraction is in A.1.

MODULE *QuasiReliable*

This module is the abstraction for a quasi-reliable channel, the primary form of communication. Communication channels connect every pair of processes and provide two basic primitives to send and receive messages. The primitives *Send* and *Receive* have the following properties:

* No creation: for p_i , p_j , if p_j invokes Received m from p_i , then p_i must have invoked *Send* m to p_j ;
* No duplication: for p_i , p_j , for all *Send* m to p_j invoked by p_i , p_j invokes a corresponding Received from p_i is at most once;
* No loss: for p_i , p_j , if process p_i invokes *Send* m to p_j , and if neither p_i nor p_j fails, then eventually Received m from p_i is invoked in p_j .

LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *Sequences*

Number of groups.
CONSTANT *NGROUPS*

Number of processes.
CONSTANT *NPROCESSES*

The set of initial messages.
CONSTANT *INITIAL_MESSAGES*

Represents the underlying network channel.
VARIABLE *QuasiReliableChannel*

A wrapper around the *Send* primitive. This procedure sends a message m to all processes in all groups. We do this instead of a single process to process to clear things up on the client side since all usages are to send messages to all participants.

$$
\begin{aligned}
&Send(m) \triangleq \\
&\quad \land\ QuasiReliableChannel' = [\\
&\qquad g \in \text{DOMAIN } QuasiReliableChannel \mapsto [\\
&\qquad\quad p \in \text{DOMAIN } QuasiReliableChannel[g] \mapsto \\
&\qquad\qquad QuasiReliableChannel[g][p] \cup \{m\}]]
\end{aligned}
$$

The receive primitive, using only this procedure, does not consume the message. We execute the callback passing the message existent in the specific process of the given group.

$Receive(g, p, Fn(\_)) \triangleq$
$\quad \land \; \exists m \in QuasiReliableChannel[g][p] : Fn(m)$

Bellow are some helper procedures built upon the *Send* and *Receive* primitives.

A wrapper to send the messages while applying a map function to the process' network buffer. We need this because we can not execute multiple operations to a variable in a single step. For example, removing and adding a message must be a single operation. In cases where we must consume and send a message in the network, we use this wrapper.

$SendMap(Fn(\_, \_)) \triangleq$
$\quad \land \quad QuasiReliableChannel' = [$
$\qquad g \in \text{DOMAIN } QuasiReliableChannel \mapsto [$
$\qquad\quad p \in \text{DOMAIN } QuasiReliableChannel[g] \mapsto$
$\qquad\qquad Fn(p, QuasiReliableChannel[g][p])]]$

This procedure causes the process in the given to consume the specific message.

$Consume(g, p, m) \triangleq$
$\quad \land \quad QuasiReliableChannel' = [$
$\qquad QuasiReliableChannel \text{ EXCEPT } ![g][p] = @ \setminus \{m\}]$

This procedure put both the *Receive* primitive with the consume procedure together. For a received message, execute the callback and removes it from the buffer.

$ReceiveAndConsume(g, p, Fn(\_)) \triangleq$
$\quad \land Receive(g, p, \text{LAMBDA } m : Fn(m) \land Consume(g, p, m))$

Initialize the algorithm with all processes in all groups with the same set of messages.

$Init \triangleq$
$\quad \land QuasiReliableChannel = [$
$\qquad g \in 1 \,..\, NGROUPS \mapsto [$
$\qquad\quad p \in 1 \,..\, NPROCESSES \mapsto INITIAL\_MESSAGES]]$

MODULE *AtomicBroadcast*

This module is the abstraction for the Atomic Broadcast, a primitive for group communication. A process can broadcast a message to its local group, where all members will deliver in the same order.

We use a sequence to maintain the same order on all processes. New messages are added to the back and removed from the front. A group has its own order within, whereas there are no ordering requirements across groups.

LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *Sequences*

Number of groups.
CONSTANT *NGROUPS*

Number of processes.
CONSTANT *NPROCESSES*

The sequences of initial messages.
CONSTANT *INITIAL_MESSAGES*

VARIABLES
    The underlying buffer that holds all the messages.
    AtomicBroadcastBuffer

Broadcast the message to the given group. We add the message at the back of every process' sequence within this group.

$$
\begin{array}{l}
ABroadcast(g, m) \triangleq \\
\quad \land AtomicBroadcastBuffer' = [\\
\quad\quad AtomicBroadcastBuffer \text{ EXCEPT } ![g] = [\\
\quad\quad\quad p \in \text{DOMAIN } AtomicBroadcastBuffer[g] \mapsto \\
\quad\quad\quad\quad Append(AtomicBroadcastBuffer[g][p], m)]]
\end{array}
$$

Deliver the message to the process in the specific group. If there is a message in the buffer, we pass it to the callback and consume it.

$$
\begin{aligned}
&ABDeliver(g,\ p,\ Fn(\_)) \triangleq \\
&\quad \land\ Len(AtomicBroadcastBuffer[g][p]) > 0 \\
&\quad \land\ Fn(Head(AtomicBroadcastBuffer[g][p])) \\
&\quad \land\ AtomicBroadcastBuffer' = [\\
&\qquad AtomicBroadcastBuffer\ \text{EXCEPT}\ ![g][p] = \\
&\qquad\quad Tail(AtomicBroadcastBuffer[g][p])]
\end{aligned}
$$

Initialize the algorithm with the configuration values. The processes within a group will have the same sequence of messages in the same order.

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land\ AtomicBroadcastBuffer = [\\
&\qquad g \in 1 \mathrel{..} NGROUPS \mapsto [\\
&\qquad\quad p \in 1 \mathrel{..} NPROCESSES \mapsto INITIAL\_MESSAGES[g]]]
\end{aligned}
$$

MODULE *GenericBroadcast*

This module is the abstraction for the Generic Broadcast, a primitive for group communication. A process can broadcast a message to a single group, and using conflict relation processes may order the delivery order.

We use a combination of sequences; each position contains a set; each set contains commuting messages. The former has an order, whereas the latter is unordered. With this approach, we have a generic delivery.

LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *Sequences*
LOCAL INSTANCE *FiniteSets*
LOCAL INSTANCE *Commons*

CONSTANT *NGROUPS*
CONSTANT *NPROCESSES*
CONSTANT *INITIAL_MESSAGES*

The conflict relation to identify commuting messages.
CONSTANT $CONFLICTR(\_, \_)$

The underlying buffer that holds all the messages.
VARIABLE *GenericBroadcastBuffer*

We consume the message in the given group. If the set in the head is empty, we remove it; we remove only m otherwise.

$$
\begin{array}{l}
\text{LOCAL } Consume(S, m) \triangleq \\
\quad \text{IF } Cardinality(Head(S)) > 1 \text{ THEN } ReplaceAt(S, 1, Head(S) \setminus \{m\}) \\
\quad \text{ELSE } SubSeq(S, 2, Len(S))
\end{array}
$$

Verify if exists conflict in the process for the message.

$$
\begin{array}{l}
\text{LOCAL } ConflictIn(V, m) \triangleq \exists \langle n, x, y \rangle \in V : CONFLICTR(m, n) \\
\text{LOCAL } HasConflict(S, m) \triangleq \\
\quad Len(SelectSeq(S, \text{LAMBDA } V : ConflictIn(V, m[1]))) \neq 0
\end{array}
$$

We insert a message to the specific process' buffer. If the buffer is empty or there is a conflict, we add the message to the back of the sequence; otherwise, we add the message in the head.

$\text{LOCAL } Insert(S, m) \triangleq$
$\quad \text{IF } Len(S) = 0 \lor HasConflict(S, m) \text{ THEN } Append(S, \{m\})$
$\quad \text{ELSE } ReplaceAt(S, Len(S), S[Len(S)] \cup \{m\})$

Broadcast a message to the given group. We insert the message in the buffer of all processes within this group.

$GBroadcast(g, m) \triangleq$
$\quad \land GenericBroadcastBuffer' = [$
$\quad\quad GenericBroadcastBuffer \text{ EXCEPT } ![g] = [$
$\quad\quad\quad i \in 1 \mathrel{..} Len(GenericBroadcastBuffer[g]) \mapsto$
$\quad\quad\quad\quad Insert(GenericBroadcastBuffer[g][i], m)]]$

Generic deliver primitive to the process in the specific group. If the buffer is not empty, we invoke the call with the appropriate message and then consume it.

$GBDeliver(g, p, Fn(\_)) \triangleq$
$\quad \land Len(GenericBroadcastBuffer[g][p]) > 0$
$\quad \land Cardinality(Head(GenericBroadcastBuffer[g][p])) > 0$
$\quad \land \text{LET}$

Since messages in the same set commute, we can choose any.

$\quad\quad m \triangleq \text{CHOOSE } v \in Head(GenericBroadcastBuffer[g][p]) : \text{TRUE}$
$\quad\quad \text{IN}$
$\quad\quad\quad \land Fn(m)$
$\quad\quad\quad \land GenericBroadcastBuffer' = [$
$\quad\quad\quad\quad GenericBroadcastBuffer \text{ EXCEPT } ![g][p] =$
$\quad\quad\quad\quad\quad Consume(GenericBroadcastBuffer[g][p], m)]$

Initialize the algorithm with the configuration values. The processes within a group will have the same sequence of messages.

$Init \triangleq$
$\quad \land GenericBroadcastBuffer = [$
$\quad\quad g \in 1 \mathrel{..} NGROUPS \mapsto [$
$\quad\quad\quad p \in 1 \mathrel{..} NPROCESSES \mapsto INITIAL\_MESSAGES[g]]]$

These are all the communication primitives. These modules are instantiated in the algorithm's modules and used as primitive.

A.2 Helper Procedures

This chapter contains the module with helper procedures and the Memory structure. The helper methods revolve around methods to help build the message structures. The Memory module is the `Mem` structure used in Generic Multicast 1 and 2.

MODULE *Commons*

LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *FiniteSets*
LOCAL INSTANCE *Sequences*

LOCAL $Identity(x) \triangleq x$
LOCAL $Choose(S) \triangleq$ CHOOSE $x \in S$: TRUE
LOCAL $IsEven(x) \triangleq x\%2 = 0$
$Max(S) \triangleq$ CHOOSE $x \in S : \forall y \in S : x \geq y$

Three different conflict relations. We identify the relation to use through the configuration files. We verify each property with all three.

Use the message's identifier, where the evens conflict with evens and odds with odds. This relationship has a partial ordering.

$IdConflict(m, n) \triangleq IsEven(m.id) = IsEven(n.id)$

All messages conflict in this relationship. The executions with this conflict relation are equivalent to the Atomic Multicast.

$AlwaysConflict(m, n) \triangleq$ TRUE

There is no conflict in this relationship. The executions with this conflict relation are equivalent to the Reliable Multicast.

$NeverConflict(m, n) \triangleq$ FALSE

We use multiple procedures provided by the TLA$^+$ community.
Most of the procedures are used locally to create the messages.

From Community Modules

LOCAL $IsInjective(f) \triangleq$

A function is injective iff it maps each element in its domain to a distinct element.

This definition is overridden by *TLC* in the *Java* class *SequencesExt*. The operator is overridden by the *Java* method with the same name.

$\forall a, b \in$ DOMAIN $f : f[a] = f[b] \Rightarrow a = b$

From Community Modules

LOCAL $SetToSeq(S) \triangleq$

Convert a set to some sequence that contains all the elements of the set exactly once, and contains no other elements.

$\text{CHOOSE } f \in [1 \,..\, Cardinality(S) \rightarrow S] : IsInjective(f)$

From Community Modules

LOCAL $SetToSeqs(S) \triangleq$

Convert the set S to a set containing all sequences containing the elements of S exactly once and no other elements. Example:

$SetToSeqs(\{\}), \{\langle\rangle\}$
$SetToSeqs(\{\text{“}t\text{”}, \text{“}l\text{”}\}) = \{\langle\text{“}t\text{”}, \text{“}l\text{”}\rangle, \langle\text{“}l\text{”}, \text{“}t\text{”}\rangle\}$

$\text{LET } D \triangleq 1 \,..\, Cardinality(S)$
$\text{IN } \{f \in [D \rightarrow S] : \forall\, i, j \in D : i \neq j \Rightarrow f[i] \neq f[j]\}$

From Community Modules

LOCAL $SetToAllKPermutations(S) \triangleq$

Convert the set S to a set containing all k-permutations of elements of S for $k \in 0 \,..\, Cardinality(S)$. Example:

$SetToAllKPermutations(\{\}) = \{\langle\rangle\}$
$SetToAllKPermutations(\{\text{“}a\text{”}\}) = \{\langle\rangle, \langle\text{“}a\text{”}\rangle\}$
$SetToAllKPermutations(\{\text{“}a\text{”}, \text{“}b\text{”}\}) =$
$\{\langle\rangle, \langle\text{“}a\text{”}\rangle, \langle\text{“}b\text{”}\rangle, \langle\text{“}a\text{”}, \text{“}b\text{”}\rangle, \langle\text{“}b\text{”}, \text{“}a\text{”}\rangle\}$

$\text{UNION } \{SetToSeqs(s) : s \in \text{SUBSET } S\}$

From Community Modules

LOCAL $MapThenFoldSet(op(\_, \_), base, f(\_), choose(\_), S) \triangleq$

Starting from base, apply op to $f(x)$, for all $x \in S$, by choosing the set elements with $choose$. If there are multiple ways for choosing an element, op should be associative and commutative. Otherwise, the result may depend on the concrete implementation of $choose$.

FoldSet, a simpler version for sets is contained in *FiniteSetsEx*. *FoldFunction*, a simpler version for functions is contained in Functions. *FoldSequence*, a simpler version for sequences is contained in *SequencesExt*.

Example:

$MapThenFoldSet(\text{LAMBDA } x, y : x \cup y,$
$\{\},$
$\text{LAMBDA } x : \{\{x\}\},$
$\text{LAMBDA } set : \text{CHOOSE } x \in set : \text{TRUE},$
$\{1, 2\})$

$= \{\{1\}, \{2\}\}$

$\text{LET } iter[s \in \text{SUBSET } S] \triangleq$
$\quad \text{IF } s = \{\} \text{ THEN } base$
$\quad \text{ELSE } \text{LET } x \triangleq choose(s)$
$\quad\quad \text{IN } op(f(x), iter[s \setminus \{x\}])$
$\text{IN } iter[S]$

From Community Modules

$\text{LOCAL } ToSet(s) \triangleq$

The image of the given sequence s. $Cardinality(ToSet(s)) \leq Len(s)$ see https://*en.wikipedia.org*/wiki/*Image_(mathematics)*

$\{s[i] : i \in \text{DOMAIN } s\}$

From Community Modules

$ReplaceAt(s, i, e) \triangleq$

Replaces the element at position i with the element e.

$[s \text{ EXCEPT } ![i] = e]$

$\text{LOCAL } Originator(G, P) \triangleq \langle Choose(G), Choose(P) \rangle$

Initialize the message structure we use to check the algorithm.

$CreateMessages(nmessage, G, P) \triangleq$
$\quad \{[id \mapsto m, d \mapsto G, o \mapsto Originator(G, P)] : m \in 1 \,..\, nmessage\}$

Create all possible different possibilities in the initial ordering. Since we replaced the combination of Reliable Multicast + Atomic Broadcast with multiple uses of Atomic Broadcast, messages can have distinct orders across groups. We force this distinction.

$CreatePossibleMessages(S) \triangleq$
$\quad \text{LET } M \triangleq SetToAllKPermutations(S)$
$\quad \text{IN } MapThenFoldSet($
$\quad\quad \text{LAMBDA } x, y : \langle x \rangle \circ y,$
$\quad\quad \langle\rangle,$
$\quad\quad Identity,$
$\quad\quad Choose,$
$\quad\quad \{m \in M : Len(m) = Cardinality(S)\})$

We create the tuple with the message, state, and timestamp.

$\text{LOCAL } InitialMessage(m) \triangleq \langle m, \text{“S0”}, 0 \rangle$

A totally ordered message buffer.

$TotallyOrdered(F) \triangleq$
$\quad [x \in \text{DOMAIN } F \mapsto InitialMessage(F[x])]$

Creates a partially ordered buffer from the sequence using the given predicate to identify conflicts between messages.

$\text{LOCAL } ExistsConflict(x, S, Op(\_, \_)) \triangleq$
$\quad \exists\, d \in ToSet(S) :$
$\quad\quad \exists\, \langle n, s, ts \rangle \in d : Op(x, n)$

$PartiallyOrdered(F, Op(\_, \_)) \triangleq$
$\quad MapThenFoldSet($
$\quad\quad \text{LAMBDA } x, y :$
$\quad\quad\quad \text{IF } Len(y) = 0 \lor ExistsConflict(x, y, Op)$
$\quad\quad\quad\quad \text{THEN } \langle \{InitialMessage(x)\} \rangle \circ y$
$\quad\quad\quad \text{ELSE } \langle y[1] \cup \{InitialMessage(x)\} \rangle,$
$\quad\quad\quad \langle\rangle,$
$\quad\quad\quad Identity,$
$\quad\quad\quad Choose,$
$\quad\quad\quad ToSet(F))$

We enumerate the entries in the given set.

$Enumerate(base, E) \triangleq$
$\quad \text{LET } f \triangleq SetToSeq(E) \text{ IN } \{\langle base + i, f[i] \rangle : i \in \text{DOMAIN } f\}$

MODULE *Memory*

This module is the abstraction for the *Memory* structure used by Generic Multicast 1 and 2. Inserting a new message will either create a new entry or update an existing one. The requirement here is that, at any time, we must always have only one entry for a message, never duplicating. Besides the insert, we have some additional procedures wrapping the buffer for verifying entries and removing them. Each process owns a buffer and accesses only its own buffer, never the others'.

LOCAL INSTANCE *FiniteSets*
LOCAL INSTANCE *Naturals*

Number of groups.
CONSTANT *NGROUPS*

Number of processes.
CONSTANT *NPROCESSES*

The underlying buffer, each process owns one.
We use a set, and the entries are the message tuples.
VARIABLE *MemoryBuffer*

Insert the new entry into the process buffer in the specific group. We remove the previous entry and put the new one in its place.

$$
\begin{aligned}
&Insert(g,\ p,\ t) \triangleq \\
&\quad \land\ MemoryBuffer' = [\\
&\quad\quad MemoryBuffer\ \text{EXCEPT}\ ![g][p] = \{ \\
&\quad\quad\quad \langle msg,\ state,\ ts\rangle \in MemoryBuffer[g][p] : \\
&\quad\quad\quad\quad msg.id \neq t[1].id\} \cup \{t\}]
\end{aligned}
$$

Verify if an entry exists in the process buffer in the specific group using the callback.

$$
\begin{aligned}
&Contains(g,\ p,\ Fn(\_)) \triangleq \\
&\quad \exists\, t \in MemoryBuffer[g][p] : Fn(t)
\end{aligned}
$$

We filter the entries in the process buffer in the specific group using the callback. An entry must be valid when compared with all others except itself.

$$
\begin{aligned}
&ForAllFilter(g,\ p,\ Fn(\_,\ \_)) \triangleq \\
&\quad \{t\_1 \in MemoryBuffer[g][p] : \\
&\quad\quad \forall\, t\_2 \in (MemoryBuffer[g][p] \setminus \{t\_1\}) : Fn(t\_1,\ t\_2)\}
\end{aligned}
$$

Remove the entries in the process buffer in the specific group.

$Remove(g,\ p,\ S) \triangleq$

$\quad \land\ MemoryBuffer' = [MemoryBuffer\ \text{EXCEPT}\ ![g][p] = @ \setminus S]$

Initialize the structure for all processes with an empty buffer.

$Init \triangleq$

$\quad \land\ MemoryBuffer = [$

$\qquad g \in 1\,..\,NGROUPS \mapsto [$

$\qquad\quad p \in 1\,..\,NPROCESSES \mapsto \{\}]]$

A.3 Generic Multicast 0

MODULE *GenericMulticast0*

LOCAL INSTANCE *Commons*
LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *FiniteSets*

Number of processes in the algorithm.
CONSTANT *NPROCESSES*

Set with initial messages the algorithm starts with.
CONSTANT *INITIAL_MESSAGES*

The conflict relation.
CONSTANT $CONFLICTR(\_, \_)$

ASSUME
  Verify that *NPROCESSES* is a natural number greater than 0.
  $\land\ NPROCESSES \in (Nat \setminus \{0\})$

  The messages in the protocol must be finite.
  $\land\ IsFiniteSet(INITIAL\_MESSAGES)$

LOCAL $Processes \triangleq \{i : i \in 1 \,..\, NPROCESSES\}$

The instance of the quasi-reliable channel for process communication primitive. We use groups with single processes, having *NPROCESSES* groups.
VARIABLE *QuasiReliableChannel*
QuasiReliable $\triangleq$ INSTANCE *QuasiReliable* WITH
  $NGROUPS \leftarrow NPROCESSES$,
  $NPROCESSES \leftarrow 1$

VARIABLES
  Structure that holds the clocks for all processes.
  K,

  Structure that holds all messages that were received but are still pending a final timestamp.
  Pending,

  Structure that holds all messages that contains a final timestamp but were not delivered yet.
  Delivering,

  Structure that holds all messages that contains a final timestamp and were already delivered.

$Delivered,$

Used to verify if previous messages conflict with the message beign processed. Using this approach is possible to deliver messages with a partially ordered delivery.

$PreviousMsgs,$

Set used to holds the votes that were cast for a message. Since the coordinator needs that all processes cast a vote for the final timestamp, this structure will hold the votes each process cast for each message on the system.

$Votes$

$vars \triangleq \langle QuasiReliableChannel, Votes, K, Pending,$
$\quad Delivering, Delivered, PreviousMsgs\rangle$

Helper to send messages. In a single operation we consume the message from our local network and send a request to the algorithm initiator. Is not possible to execute multiple operations in a single step on the same set. That is, we can not consume and send in different operations.

$\text{LOCAL } SendOriginatorAndRemoveLocal(self, dest, curr, prev, S) \triangleq$
$\quad \text{IF } self = dest \land prev[2].o = self \text{ THEN } (S \setminus \{prev\}) \cup \{curr\}$
$\quad\ \text{ELSE IF } prev[2].o = dest \text{ THEN } S \cup \{curr\}$
$\quad\ \text{ELSE IF } self = dest \text{ THEN } S \setminus \{prev\}$
$\quad\ \text{ELSE } S$

Check if the given message conflict with any other in the *PreviousMsgs*.

$\text{LOCAL } HasConflict(self, m1) \triangleq$
$\quad \exists\, m2 \in PreviousMsgs[self] : CONFLICTR(m1, m2)$

We have the handlers representing each step of the algorithm. The handlers are the actual algorithm, and the caller is the step guard predicate.

$\text{LOCAL } AssignTimestampHandler(self, msg) \triangleq$
$\quad \land \lor \land HasConflict(self, msg)$
$\qquad\quad \land K' = [K \text{ EXCEPT } ![self] = K[self] + 1]$
$\qquad\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![self] = \{msg\}]$
$\qquad \lor \land \neg HasConflict(self, msg)$
$\qquad\quad \land K' = [K \text{ EXCEPT } ![self] = K[self]]$
$\qquad\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![self] =$
$\qquad\qquad\qquad PreviousMsgs[self] \cup \{msg\}]$
$\quad \land Pending' = [Pending \text{ EXCEPT } ![self] = Pending[self] \cup \{\langle K'[self], msg\rangle\}]$
$\quad \land QuasiReliable!SendMap(\text{LAMBDA } dest, S :$
$\qquad SendOriginatorAndRemoveLocal(self, dest,$
$\qquad\quad \langle \text{“S1”}, K'[self], msg, self\rangle, \langle \text{“S0”}, msg\rangle, S))$
$\quad \land \text{UNCHANGED } \langle Delivering, Delivered, Votes\rangle$

$$
\begin{array}{l}
\text{LOCAL } \mathit{ComputeSeqNumberHandler}(\mathit{self}, \mathit{ts}, \mathit{msg}, \mathit{origin}) \triangleq \\
\quad \land \text{ LET} \\
\qquad \mathit{vote} \triangleq \langle \mathit{msg.id}, \mathit{origin}, \mathit{ts} \rangle \\
\qquad \mathit{election} \triangleq \{v \in (\mathit{Votes}[\mathit{self}] \cup \{\mathit{vote}\}) : v[1] = \mathit{msg.id}\} \\
\qquad \mathit{elected} \triangleq \mathit{Max}(\{x[3] : x \in \mathit{election}\}) \\
\quad\;\; \text{IN} \\
\qquad \land \lor \land \mathit{Cardinality}(\mathit{election}) = \mathit{Cardinality}(\mathit{msg.d}) \\
\qquad\qquad \land \mathit{Votes}' = [\mathit{Votes} \text{ EXCEPT } ![\mathit{self}] = \\
\qquad\qquad\qquad \{x \in \mathit{Votes}[\mathit{self}] : x[1] \neq \mathit{msg.id}\}] \\
\qquad\qquad \land \mathit{QuasiReliable}!\mathit{SendMap}(\text{LAMBDA } \mathit{dest}, S : \\
\qquad\qquad\qquad (S \setminus \{\langle \text{“S1”}, \mathit{ts}, \mathit{msg} \rangle\}) \cup \{\langle \text{“S2”}, \mathit{elected}, \mathit{msg} \rangle\}) \\
\qquad\quad \lor \land \mathit{Cardinality}(\mathit{election}) < \mathit{Cardinality}(\mathit{msg.d}) \\
\qquad\qquad \land \mathit{Votes}' = [\mathit{Votes} \text{ EXCEPT } ![\mathit{self}] = \mathit{Votes}[\mathit{self}] \cup \{\mathit{vote}\}] \\
\qquad\qquad \land \mathit{QuasiReliable}!\mathit{Consume}(1, \mathit{self}, \langle \text{“S1”}, \mathit{ts}, \mathit{msg}, \mathit{origin} \rangle) \\
\qquad \land \text{ UNCHANGED } \langle K, \mathit{PreviousMsgs}, \mathit{Pending}, \mathit{Delivering}, \mathit{Delivered} \rangle
\end{array}
$$

$$
\begin{array}{l}
\text{LOCAL } \mathit{AssignSeqNumberHandler}(\mathit{self}, \mathit{ts}, \mathit{msg}) \triangleq \\
\quad \land \lor \land \mathit{ts} > K[\mathit{self}] \\
\qquad\quad \land \lor \land \mathit{HasConflict}(\mathit{self}, \mathit{msg}) \\
\qquad\qquad\quad \land K' = [K \text{ EXCEPT } ![\mathit{self}] = \mathit{ts} + 1] \\
\qquad\qquad\quad \land \mathit{PreviousMsgs}' = [\mathit{PreviousMsgs} \text{ EXCEPT } ![\mathit{self}] = \{\}] \\
\qquad\qquad \lor \land \neg \mathit{HasConflict}(\mathit{self}, \mathit{msg}) \\
\qquad\qquad\quad \land K' = [K \text{ EXCEPT } ![\mathit{self}] = \mathit{ts}] \\
\qquad\qquad\quad \land \text{ UNCHANGED } \mathit{PreviousMsgs} \\
\qquad \lor \land \mathit{ts} \leq K[\mathit{self}] \\
\qquad\quad \land \text{ UNCHANGED } \langle K, \mathit{PreviousMsgs} \rangle \\
\quad \land \mathit{Delivering}' = [\mathit{Delivering} \text{ EXCEPT } ![\mathit{self}] = \mathit{Delivering}[\mathit{self}] \cup \{\langle \mathit{ts}, \mathit{msg} \rangle\}] \\
\quad \land \text{ UNCHANGED } \langle \mathit{Votes}, \mathit{Delivered} \rangle
\end{array}
$$

This procedure executes after an initiator GM-Cast a message m to $m.d$. All processes in $m.d$ do the same thing after receiving m, assing the local clock to the message timestamp, inserting the message with the timestamp to the process *Pending* set, and sending it to the initiator to choose the timestamp.

$$
\begin{array}{l}
\mathit{AssignTimestamp}(\mathit{self}) \triangleq \\
\quad \text{We delegate to the lambda to handle the message while filtering for} \\
\quad \text{the correct state.} \\
\quad \land \mathit{QuasiReliable}!\mathit{Receive}(\mathit{self}, 1, \\
\qquad \text{LAMBDA } t : \\
\qquad\quad \land \quad t[1] = \text{“S0”} \\
\qquad\quad \land \quad \mathit{AssignTimestampHandler}(\mathit{self}, t[2]))
\end{array}
$$

This method is executed only by the initiator. This method processes messages on state $S1$ and can proceed in two ways. If the initiator has votes from all other processes, the message's final timestamp is the maximum received vote, and the initiator sends the message back to all participants in state $S2$. Otherwise, the initiator only store the received message in the *Votes* structure.

$ComputeSeqNumber(self) \triangleq$

We delegate to the lambda handler to effectively execute the procedure. Here we verify that the message is on state $S1$ and the current process is the initiator.

$\land$ *QuasiReliable* ! *Receive*(*self*, 1,
LAMBDA t :
$\land$ $t[1] =$ "S1"
$\land$ $t[3].o = self$
$\land$ *ComputeSeqNumberHandler*(*self*, $t[2]$, $t[3]$, $t[4]$))

After the coordinator computes the final timestamp for the message m, all processes in $m.d$ will receive the chosen timestamp. Each participant checks the message's timestamp against its local clock. If the value is greater than the process clock, we need to update the process clock with the message's timestamp. If m conflicts with a message in the *PreviousMsgs*, the clock updates to m's timestamp plus one and clears the *PreviousMsgs* set. Without any conflict with m, the clock updates to m's timestamp. The message is removed from *Pending* and added to *Delivering* set.

$AssignSeqNumber(self) \triangleq$

We delegate the procedure execution the the handler, and the message is automatically consumed after the lambda execution. In this one we only filter the messages.

$\land$ *QuasiReliable* ! *ReceiveAndConsume*(*self*, 1,
LAMBDA t_1 :
$\land$ $t\_1[1] =$ "S2"
$\land$ $\exists\, t\_2 \in Pending[self] : t\_1[3].id = t\_2[2].id$
$\land$ *AssignSeqNumberHandler*(*self*, $t\_1[2]$, $t\_1[3]$)
We remove the message here to avoid too many arguments in the procedure invocation.
$\land$ $Pending' = [Pending$ EXCEPT $![self] = @ \setminus \{t\_2\}])$

Responsible for delivery of messages. The messages in the *Delivering* set with the smallest timestamp among others in the *Pending* joined with *Delivering* set. We can also deliver messages that commute with all others, the generalized behavior in action.

Delivered messages will be added to the *Delivered* set and removed from the others. To store the instant of delivery, we insert delivered messages with the following format:

```
<<Nat, Message>>
```

Using this model, we know the message delivery order for all processes.

$$
\begin{array}{l}
\mathit{DoDeliver}(\mathit{self}) \triangleq \\
\quad \exists \langle \mathit{ts\_1}, \mathit{m\_1} \rangle \in \mathit{Delivering}[\mathit{self}] : \\
\quad\quad \land \forall \langle \mathit{ts\_2}, \mathit{m\_2} \rangle \in (\mathit{Delivering}[\mathit{self}] \cup \mathit{Pending}[\mathit{self}]) \setminus \{\langle \mathit{ts\_1}, \mathit{m\_1} \rangle\} : \\
\quad\quad\quad \lor \neg \mathit{CONFLICTR}(\mathit{m\_1}, \mathit{m\_2}) \\
\quad\quad\quad \lor \mathit{ts\_1} < \mathit{ts\_2} \lor (\mathit{m\_1}.\mathit{id} < \mathit{m\_2}.\mathit{id} \land \mathit{ts\_1} = \mathit{ts\_2}) \\
\quad\quad \land \text{LET} \\
\quad\quad\quad T \triangleq \mathit{Delivering}[\mathit{self}] \cup \mathit{Pending}[\mathit{self}] \\
\quad\quad\quad G \triangleq \{\mathit{t\_i} \in \mathit{Delivering}[\mathit{self}] : \\
\quad\quad\quad\quad \forall \mathit{t\_j} \in T \setminus \{\mathit{t\_i}\} : \neg \mathit{CONFLICTR}(\mathit{t\_i}[2], \mathit{t\_j}[2])\} \\
\quad\quad\quad F \triangleq \{\mathit{m\_1}\} \cup \{t[2] : t \in G\} \\
\quad\quad\quad \text{IN} \\
\quad\quad\quad \land \mathit{Delivering}' = [\mathit{Delivering} \text{ EXCEPT } ![\mathit{self}] = @ \setminus (G \cup \{\langle \mathit{ts\_1}, \mathit{m\_1} \rangle\})] \\
\quad\quad\quad \land \mathit{Delivered}' = [\mathit{Delivered} \text{ EXCEPT } ![\mathit{self}] = \\
\quad\quad\quad\quad\quad \mathit{Delivered}[\mathit{self}] \cup \mathit{Enumerate}(\mathit{Cardinality}(\mathit{Delivered}[\mathit{self}]), F)] \\
\quad\quad\quad \land \text{UNCHANGED } \langle \mathit{QuasiReliableChannel}, \mathit{Votes}, \mathit{Pending}, \mathit{PreviousMsgs}, K \rangle
\end{array}
$$

Responsible for initializing global variables used on the system. All variables necessary by the protocol are a mapping from the node *id* to the corresponding process set.

The "message" is also a structure, with the following format:

```
[ id |-> Nat, d |-> Nodes, o |-> Node ]
```

We have the properties: *id* is the messages' unique *id*, we use a natural number to represent; *d* is the destination, it may be a subset of the Nodes set; and *o* is the originator, the process that started the execution of the algorithm. These properties are all static and never change.

The mutable values we transport outside the message structure. We do this using the process communication channel, using a tuple to send the message along with the mutable values.

$$
\begin{array}{l}
\text{LOCAL } \mathit{InitProtocol} \triangleq \\
\quad \land K = [i \in \mathit{Processes} \mapsto 0] \\
\quad \land \mathit{Pending} = [i \in \mathit{Processes} \mapsto \{\}] \\
\quad \land \mathit{Delivering} = [i \in \mathit{Processes} \mapsto \{\}] \\
\quad \land \mathit{Delivered} = [i \in \mathit{Processes} \mapsto \{\}] \\
\quad \land \mathit{PreviousMsgs} = [i \in \mathit{Processes} \mapsto \{\}]
\end{array}
$$

LOCAL $InitHelpers \triangleq$

Initialize the protocol network.

$\land\ QuasiReliable!Init$

This structure is holding the votes the processes cast for each message on the system. Since any process can be the "coordinator", this is a mapping for processes to a set. The set will contain the vote a process has cast for a message.

$\land\ Votes = [i \in Processes \mapsto \{\}]$

$Init \triangleq InitProtocol \land InitHelpers$

$Step(self) \triangleq$

$\lor\ AssignTimestamp(self)$
$\lor\ ComputeSeqNumber(self)$
$\lor\ AssignSeqNumber(self)$
$\lor\ DoDeliver(self)$

$Next \triangleq$

$\lor\ \exists\, self \in Processes : Step(self)$
$\lor$ UNCHANGED $vars$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$SpecFair \triangleq Spec \land \mathrm{WF}_{vars}(\exists\, self \in Processes : Step(self))$

Helper functions to aid when checking the algorithm properties.

$WasDelivered(p, m) \triangleq$

Verifies if the given process p delivered message m.

$\land\ \exists\, \langle idx, n\rangle \in Delivered[p] : n.id = m.id$

$DeliveredInstant(p, m) \triangleq$

Retrieve the instant the given process p delivered message m.

$(\text{CHOOSE } \langle index, n\rangle \in Delivered[p] : m.id = n.id)[1]$

$FilterDeliveredMessages(p, m) \triangleq$

Retrieve the set of messages with the same id as message m delivered by the given process p.

$\{\langle idx, n\rangle \in Delivered[p] : n.id = m.id\}$

A.4 Generic Multicast 1

```
──────────────── MODULE GenericMulticast1 ────────────────
LOCAL INSTANCE Commons
LOCAL INSTANCE Naturals
LOCAL INSTANCE FiniteSets
LOCAL INSTANCE TLC

  Number of groups in the algorithm.
CONSTANT NGROUPS

  Number of processes in the algorithm.
CONSTANT NPROCESSES

  Set with initial messages the algorithm starts with.
CONSTANT INITIAL_MESSAGES

  The conflict relation.
CONSTANT CONFLICTR(_, _)
──────────────────────────────────────────────────────────

ASSUME
    Verify that NGROUPS is a natural number greater than 0.
  ∧ NGROUPS ∈ (Nat \ {0})
    Verify that NPROCESSES is a natural number greater than 0.
  ∧ NPROCESSES ∈ (Nat \ {0})
──────────────────────────────────────────────────────────

LOCAL Processes ≜ {p : p ∈ 1 .. NPROCESSES}
LOCAL Groups ≜ {g : g ∈ 1 .. NGROUPS}

  The module containing the Atomic Broadcast primitive.
VARIABLE AtomicBroadcastBuffer
AtomicBroadcast ≜ INSTANCE AtomicBroadcast

  The module containing the quasi reliable channel.
VARIABLE QuasiReliableChannel
QuasiReliable ≜ INSTANCE QuasiReliable WITH
   INITIAL_MESSAGES ← {}

  The algorithm's Mem structure. We use a separate module.
VARIABLE MemoryBuffer
Memory ≜ INSTANCE Memory
──────────────────────────────────────────────────────────

VARIABLES
    The process local clock.
```

K,

The set contains previous messages. We use this with the *CONFLICTR* to verify conflicting messages.

$PreviousMsgs$,

The set of delivered messages. This set is not an algorithm requirement. We use this to help check the algorithm's properties.

$Delivered$,

A set contains the processes' votes for the message's timestamp. This structure is implicit in the algorithm.

$Votes$

$$
\begin{array}{l}
vars \triangleq \langle \\
\quad K, \\
\quad MemoryBuffer, \\
\quad PreviousMsgs, \\
\quad Delivered, \\
\quad Votes, \\
\quad AtomicBroadcastBuffer, \\
\quad QuasiReliableChannel \\
\rangle
\end{array}
$$

Check if the given message conflict with any other in the *PreviousMsgs*.

$$
\begin{array}{l}
\textsc{local}\ HasConflict(g, p, m1) \triangleq \\
\quad \exists\, m2 \in PreviousMsgs[g][p] : CONFLICTR(m1, m2)
\end{array}
$$

These are the handlers. The actual algorithm resides here, the lambdas only assert the guarding predicates before calling the handler.

$$
\begin{array}{l}
\textsc{local}\ ComputeGroupSeqNumberHandler(g, p, msg, ts) \triangleq \\
\quad \land \lor \land HasConflict(g, p, msg) \\
\qquad\quad \land K' = [K \ \textsc{except}\ ![g][p] = K[g][p] + 1] \\
\qquad\quad \land PreviousMsgs' = [PreviousMsgs \ \textsc{except}\ ![g][p] = \{msg\}] \\
\qquad \lor \land \neg HasConflict(g, p, msg) \\
\qquad\quad \land PreviousMsgs' = [PreviousMsgs \ \textsc{except}\ ![g][p] = \\
\qquad\qquad\quad PreviousMsgs[g][p] \cup \{msg\}] \\
\qquad\quad \land \textsc{unchanged}\ K \\
\quad \land \lor \land Cardinality(msg.d) > 1 \\
\qquad\quad \land Memory!Insert(g, p, \langle msg, \text{"S1"}, K'[g][p]\rangle) \\
\qquad\quad \land QuasiReliable!Send(\langle msg, g, K'[g][p]\rangle) \\
\qquad \lor \land Cardinality(msg.d) = 1 \\
\qquad\quad \land Memory!Insert(g, p, \langle msg, \text{"S3"}, K'[g][p]\rangle) \\
\qquad\quad \land \textsc{unchanged}\ QuasiReliableChannel \\
\quad \land \textsc{unchanged}\ \langle Delivered, Votes\rangle
\end{array}
$$

$\text{LOCAL } SynchronizeGroupClockHandler(g, p, m, tsf) \triangleq$
$\land \lor \land tsf > K[g][p]$
$\quad \land K' = [K \text{ EXCEPT } ![g][p] = tsf]$
$\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![g][p] = \{\}]$
$\lor \land tsf \leq K[g][p]$
$\quad \land \text{UNCHANGED } \langle K, PreviousMsgs \rangle$
$\land \lor \land \exists \langle n, s, ts \rangle \in MemoryBuffer[g][p] : s = \text{"S1"} \land m = n$
$\quad \land Memory!Insert(g, p, \langle m, \text{"S3"}, tsf \rangle)$
$\lor \text{UNCHANGED } MemoryBuffer$
$\land \text{UNCHANGED } \langle QuasiReliableChannel, Delivered, Votes \rangle$

$\text{LOCAL } GatherGroupsTimestampHandler(g, p, msg, ts, tsf) \triangleq$
$\land \quad \lor \land ts < tsf$
$\quad \land AtomicBroadcast!ABroadcast(g, \langle msg, \text{"S2"}, tsf \rangle)$
$\lor \text{UNCHANGED } AtomicBroadcastBuffer$
$\land \quad Memory!Insert(g, p, \langle msg, \text{"S3"}, tsf \rangle)$
$\land \quad \text{UNCHANGED } \langle K, PreviousMsgs, Delivered \rangle$

Executes when process P receives a message M from the Atomic Broadcast primitive and M is in P's memory. This procedure is extensive, with multiple branches based on the message's state and destination. Let's split the explanation.

When M's state is $S0$, we first verify if M conflicts with messages in the *PreviousMsgs* set. If a conflict exists, we increase P's local clock by one and clear the *PreviousMsgs* set.

When message M has a single group as the destination, it is already in its desired destination and is synchronized because we received M from Atomic Broadcast primitive. P stores M in memory with state $S3$ and timestamp with the current clock value.

When M includes multiple groups in the destination, the participants must agree on the final timestamp. When M's state is $S0$, P will send its timestamp proposition to all other participants, which is the current clock value, and update M's state to $S1$ and timestamp. If M's state is $S2$, we are synchronizing the local group, meaning we may need to leap the clock to the M's received timestamp and then set M to state $S3$.

$ComputeGroupSeqNumber(g, p) \triangleq$
$\land AtomicBroadcast!ABDeliver(g, p,$
$\quad \text{LAMBDA } t : t[2] = \text{"S0"} \land ComputeGroupSeqNumberHandler(g, p, t[1], t[3]))$

After exchanging the votes between groups, processes must select the final timestamp. When we have one proposal from each group in message M's destination, the highest vote is the decided timestamp. If P's local clock is smaller than the value, we broadcast the message to the local group with state $S2$ and save it in memory. Otherwise, we update the in-memory to state $S3$.

We only execute the procedure once we have proposals from all participating groups. Since we receive messages from the quasi-reliable channel, we keep the votes in the *Votes* structure. This structure is implicit in the algorithm.

$\text{LOCAL } HasNecessaryVotes(g, p, msg, ballot) \triangleq$
$\land Cardinality(ballot) = Cardinality(msg.d)$
$\land Memory!Contains(g, p, \text{LAMBDA } n : msg = n[1] \land n[2] = \text{"S1"})$

```
GatherGroupsTimestamp(g, p) ≜
    ∧ QuasiReliable!ReceiveAndConsume(g, p,
        LAMBDA t :
            ∧   LET
                    msg ≜ t[1]
                    origin ≜ t[2]
                    vote ≜ ⟨msg.id, origin, t[3]⟩
                    ballot ≜ {v ∈ (Votes[g][p] ∪ {vote}) : v[1] = msg.id}
                    elected ≜ Max({x[3] : x ∈ ballot})
                IN
                    We only execute the procedure when we have proposals from all groups.
                    ∧ ∨ ∧ HasNecessaryVotes(g, p, msg, ballot)
                        ∧ ∃ ⟨m, s, ts⟩ ∈ MemoryBuffer[g][p] : m = msg
                          ∧ GatherGroupsTimestampHandler(g, p, msg, ts, elected)
                        ∧ Votes' = [Votes EXCEPT ![g][p] = {
                            x ∈ Votes[g][p] : x[1] ≠ msg.id}]
                      ∨ ∧ ¬HasNecessaryVotes(g, p, msg, ballot)
                        ∧ Votes' = [Votes EXCEPT ![g][p] = Votes[g][p] ∪ {vote}]
                        ∧ UNCHANGED ⟨MemoryBuffer, K,
                            PreviousMsgs, AtomicBroadcastBuffer⟩
                    ∧ UNCHANGED ⟨Delivered⟩)

SynchronizeGroupClock(g, p) ≜
    ∧ AtomicBroadcast!ABDeliver(g, p,
        LAMBDA t : t[2] = "S2" ∧ SynchronizeGroupClockHandler(g, p, t[1], t[3]))
```

When messages are to deliver, we select them and call the delivery primitive. Ready means they are in state $S3$, and the message either does not conflict with any other in the memory structure or is smaller than all others. Once a message is ready, we also collect the messages that do not conflict with any other for delivery in a single batch.

```
DoDeliver(g, p) ≜
    We are accessing the buffer directly, and not through the Memory instance.
    We do this because is easier and because we are only reading the values here.
    Any changes we do through the instance.
    ∃ ⟨m_1, state, ts_1⟩ ∈ MemoryBuffer[g][p] :
      ∧ state = "S3"
      ∧ ∀ ⟨m_2, ignore, ts_2⟩ ∈ (MemoryBuffer[g][p] \ {⟨m_1, state, ts_1⟩}) :
          ∧ ∨ ¬CONFLICTR(m_1, m_2)
            ∨ ts_1 < ts_2 ∨ (m_1.id < m_2.id ∧ ts_1 = ts_2)
      ∧ LET
          G ≜ Memory!ForAllFilter(g, p,
                LAMBDA t_i, t_j : t_i[2] = "S3" ∧ ¬CONFLICTR(t_i[1], t_j[1]))
          D ≜ G ∪ {⟨m_1, "S3", ts_1⟩}
          F ≜ {t[1] : t ∈ D}
        IN
          ∧ Memory!Remove(g, p, D)
```

$$\wedge\ Delivered' = [Delivered\ \text{EXCEPT}\ ![g][p] = Delivered[g][p] \cup Enumerate(Cardinality(Delivered[g][p]),\ F)]$$
$$\wedge\ \text{UNCHANGED}\ \langle QuasiReliableChannel,\ AtomicBroadcastBuffer,\ Votes,\ PreviousMsgs,\ K\rangle$$

$\text{LOCAL}\ InitProtocol \triangleq$
$\wedge\ K = [i \in Groups \mapsto [p \in Processes \mapsto i]]$
$\wedge\ Memory!Init$
$\wedge\ PreviousMsgs = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$
$\wedge\ Delivered = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$
$\wedge\ Votes = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$

$\text{LOCAL}\ InitCommunication \triangleq$
$\wedge\ AtomicBroadcast!Init$
$\wedge\ QuasiReliable!Init$

$Init \triangleq InitProtocol \wedge InitCommunication$

$Step(g,\ p) \triangleq$
$\vee\ ComputeGroupSeqNumber(g,\ p)$
$\vee\ GatherGroupsTimestamp(g,\ p)$
$\vee\ SynchronizeGroupClock(g,\ p)$
$\vee\ DoDeliver(g,\ p)$

$GroupStep(g) \triangleq$
$\exists\, p \in Processes : Step(g,\ p)$

$Next \triangleq$
$\vee\ \exists\, g \in Groups : GroupStep(g)$
$\vee\ \text{UNCHANGED}\ vars$

$Spec \triangleq Init \wedge \Box[Next]_{vars}$

$SpecFair \triangleq Spec \wedge \text{WF}_{vars}(\exists\, g \in Groups : GroupStep(g))$

Helper functions to aid when checking the algorithm properties.

$WasDelivered(g,\ p,\ m) \triangleq$

Verifies if the given process p in group g delivered message m .

$\wedge\ \exists\, \langle idx,\ n\rangle \in Delivered[g][p] : n.id = m.id$

$FilterDeliveredMessages(g,\ p,\ m) \triangleq$

Retrieve the set of messages with the same id as message m delivered by the given process p in group g .

$\{\langle idx, n\rangle \in Delivered[g][p] : n.id = m.id\}$

$DeliveredInstant(g, p, m) \triangleq$

Retrieve the instant the process p in group g delivered message m .

$(\text{CHOOSE } \langle t, n\rangle \in Delivered[g][p] : n.id = m.id)[1]$

A.5 Generic Multicast 2

MODULE *GenericMulticast2*

LOCAL INSTANCE *Commons*
LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *FiniteSets*

Number of groups in the algorithm.
CONSTANT *NGROUPS*

Number of processes in the algorithm.
CONSTANT *NPROCESSES*

Set with initial messages the algorithm starts with.
CONSTANT *INITIAL_MESSAGES*

The conflict relation.
CONSTANT $CONFLICTR(\_, \_)$

ASSUME
  Verify that *NGROUPS* is a natural number greater than 0.
  $\land\ NGROUPS \in (Nat \setminus \{0\})$
  Verify that *NPROCESSES* is a natural number greater than 0.
  $\land\ NPROCESSES \in (Nat \setminus \{0\})$

LOCAL $Processes \triangleq \{p : p \in 1 \mathrel{..} NPROCESSES\}$
LOCAL $Groups \triangleq \{g : g \in 1 \mathrel{..} NGROUPS\}$

The module containing the Generic Broadcast primitive.
VARIABLE *GenericBroadcastBuffer*
GenericBroadcast $\triangleq$ INSTANCE *GenericBroadcast*

The module containing the quasi reliable channel.
VARIABLE *QuasiReliableChannel*
QuasiReliable $\triangleq$ INSTANCE *QuasiReliable* WITH
  $INITIAL\_MESSAGES \leftarrow \{\}$

The algorithm's *Mem* structure. We use a separate module.
VARIABLE *MemoryBuffer*
Memory $\triangleq$ INSTANCE *Memory*

VARIABLES
  The process local clock.
  K,

The set contains previous messages. We use this with the *CONFLICTR* to verify conflicting messages.

$PreviousMsgs,$

The set of delivered messages. This set is not an algorithm requirement. We use this to help check the algorithm's properties.

$Delivered,$

A set contains the processes' votes for the message's timestamp. This structure is implicit in the algorithm.

$Votes$

$vars \triangleq \langle$
$\quad K,$
$\quad MemoryBuffer,$
$\quad PreviousMsgs,$
$\quad Delivered,$
$\quad Votes,$
$\quad GenericBroadcastBuffer,$
$\quad QuasiReliableChannel$
$\rangle$

These are the handlers. The actual algorithm resides here, the lambdas only assert the guarding predicates before calling the handler.

Check if the given message conflict with any other in the *PreviousMsgs*.

$\text{LOCAL } HasConflict(g, p, m1) \triangleq$
$\quad \exists\, m2 \in PreviousMsgs[g][p] : CONFLICTR(m1, m2)$

$\text{LOCAL } ComputeGroupSeqNumberHandler(g, p, msg, ts) \triangleq$
$\quad \land \lor \land HasConflict(g, p, msg)$
$\qquad\quad \land K' = [K \text{ EXCEPT } ![g][p] = K[g][p] + 1]$
$\qquad\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![g][p] = \{msg\}]$
$\qquad \lor \land \neg HasConflict(g, p, msg)$
$\qquad\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![g][p] =$
$\qquad\qquad\quad PreviousMsgs[g][p] \cup \{msg\}]$
$\qquad\quad \land \text{UNCHANGED } K$
$\quad \land \lor \land Cardinality(msg.d) > 1$
$\qquad\quad \land Memory!Insert(g, p, \langle msg, \text{"S1"}, K'[g][p]\rangle)$
$\qquad\quad \land QuasiReliable!Send(\langle msg, g, K'[g][p]\rangle)$
$\qquad \lor \land Cardinality(msg.d) = 1$
$\qquad\quad \land Memory!Insert(g, p, \langle msg, \text{"S3"}, K'[g][p]\rangle)$
$\qquad\quad \land \text{UNCHANGED } QuasiReliableChannel$
$\quad \land \text{UNCHANGED } \langle Delivered, Votes\rangle$

$\text{LOCAL } SynchronizeGroupClockHandler(g, p, m, tsf) \triangleq$

$$
\begin{array}{l}
\land\ \lor\ \land\ tsf > K[g][p] \\
\qquad\quad \land\ K' = [K\ \text{EXCEPT}\ ![g][p] = tsf] \\
\qquad\quad \land\ PreviousMsgs' = [PreviousMsgs\ \text{EXCEPT}\ ![g][p] = \{\}] \\
\quad\ \ \lor\ \land\ tsf \leq K[g][p] \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle K,\ PreviousMsgs\rangle \\
\land\ \lor\ \land\ \exists\, \langle n,\ s,\ ts\rangle \in MemoryBuffer[g][p]: \\
\qquad\qquad \land\ s\ = \text{“S1”} \\
\qquad\qquad \land\ m = n \\
\qquad\qquad \land\ Memory!Insert(g,\ p,\ \langle m,\ \text{“S3”},\ K'[g][p]\rangle) \\
\quad\ \ \lor\ \land\ \text{UNCHANGED}\ MemoryBuffer \\
\land\ \text{UNCHANGED}\ \langle QuasiReliableChannel,\ Delivered,\ Votes\rangle
\end{array}
$$

$$
\begin{array}{l}
\text{LOCAL}\ GatherGroupsTimestampHandler(g,\ p,\ msg,\ ts,\ tsf) \triangleq \\
\quad \land\quad \lor\ \land\ ts < tsf \\
\qquad\qquad\quad \land\ GenericBroadcast!GBroadcast(g,\ \langle msg,\ \text{“S2”},\ tsf\rangle) \\
\qquad\quad\ \ \lor\ \text{UNCHANGED}\ GenericBroadcastBuffer \\
\quad \land\quad Memory!Insert(g,\ p,\ \langle msg,\ \text{“S3”},\ tsf\rangle) \\
\quad \land\quad \text{UNCHANGED}\ \langle K,\ PreviousMsgs,\ Delivered\rangle
\end{array}
$$

Executes when process P receives a message M from the Atomic Broadcast primitive and M is in P's memory. This procedure is extensive, with multiple branches based on the message's state and destination. Let's split the explanation.

When M's state is $S0$, we first verify if M conflicts with messages in the *PreviousMsgs* set. If a conflict exists, we increase P's local clock by one and clear the *PreviousMsgs* set.

When message M has a single group as the destination, it is already in its desired destination and is synchronized because we received M from Atomic Broadcast primitive. P stores M in memory with state $S3$ and timestamp with the current clock value.

When M includes multiple groups in the destination, the participants must agree on the final timestamp. When M's state is $S0$, P will send its timestamp proposition to all other participants, which is the current clock value, and update M's state to $S1$ and timestamp. If M's state is $S2$, we are synchronizing the local group, meaning we may need to leap the clock to the M's received timestamp and then set M to state $S3$.

$$
\begin{array}{l}
ComputeGroupSeqNumber(g,\ p) \triangleq \\
\quad \land\ GenericBroadcast!GBDeliver(g,\ p, \\
\qquad\quad \text{LAMBDA}\ t : t[2] = \text{“S0”} \land ComputeGroupSeqNumberHandler(g,\ p,\ t[1],\ t[3]))
\end{array}
$$

After exchanging the votes between groups, processes must select the final timestamp. When we have one proposal from each group in message M's destination, the highest vote is the decided timestamp. If P's local clock is smaller than the value, we broadcast the message to the local group with state $S2$ and save it in memory. Otherwise, we update the in-memory to state $S3$.

We only execute the procedure once we have proposals from all participating groups. Since we receive messages from the quasi-reliable channel, we keep the votes in the *Votes* structure. This structure is implicit in the algorithm.

$$
\begin{array}{l}
\text{LOCAL}\ HasNecessaryVotes(g,\ p,\ msg,\ ballot) \triangleq \\
\quad \land\ Cardinality(ballot) = Cardinality(msg.d)
\end{array}
$$

$$\begin{array}{l}
\quad \land\ Memory!Contains(g,\ p,\ \text{LAMBDA}\ n : msg.id = n[1].id \land n[2] = \text{“S1”})\\
GatherGroupsTimestamp(g,\ p) \triangleq\\
\quad \land\ QuasiReliable!ReceiveAndConsume(g,\ p,\\
\qquad \text{LAMBDA}\ t :\\
\qquad\quad \land\quad \text{LET}\\
\qquad\qquad\quad msg \triangleq t[1]\\
\qquad\qquad\quad origin \triangleq t[2]\\
\qquad\qquad\quad vote \triangleq \langle msg.id,\ origin,\ t[3]\rangle\\
\qquad\qquad\quad ballot \triangleq \{v \in (Votes[g][p] \cup \{vote\}) : v[1] = msg.id\}\\
\qquad\qquad\quad elected \triangleq Max(\{x[3] : x \in ballot\})\\
\qquad\qquad\ \ \text{IN}
\end{array}$$

We only execute the procedure when we have proposals from all groups.

$$\begin{array}{l}
\qquad\qquad\quad \land\ \lor\ \land\ HasNecessaryVotes(g,\ p,\ msg,\ ballot)\\
\qquad\qquad\qquad\qquad \land\ \exists\, \langle m,\ s,\ ts\rangle \in MemoryBuffer[g][p] : m = msg\\
\qquad\qquad\qquad\qquad\quad \land\ GatherGroupsTimestampHandler(g,\ p,\ msg,\ ts,\ elected)\\
\qquad\qquad\qquad\qquad \land\ Votes' = [Votes\ \text{EXCEPT}\ ![g][p] = \{\\
\qquad\qquad\qquad\qquad\qquad x \in Votes[g][p] : x[1] \neq msg.id\}]\\
\qquad\qquad\qquad\quad \lor\ \land\ \neg HasNecessaryVotes(g,\ p,\ msg,\ ballot)\\
\qquad\qquad\qquad\qquad \land\ Votes' = [Votes\ \text{EXCEPT}\ ![g][p] = Votes[g][p] \cup \{vote\}]\\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED}\ \langle MemoryBuffer,\ K,\\
\qquad\qquad\qquad\qquad\qquad PreviousMsgs,\ GenericBroadcastBuffer\rangle\\
\qquad\qquad\quad \land\ \text{UNCHANGED}\ \langle Delivered\rangle)
\end{array}$$

$$\begin{array}{l}
SynchronizeGroupClock(g,\ p) \triangleq\\
\quad \land\ GenericBroadcast!GBDeliver(g,\ p,\\
\qquad \text{LAMBDA}\ t : t[2] = \text{“S2”} \land SynchronizeGroupClockHandler(g,\ p,\ t[1],\ t[3]))
\end{array}$$

When messages are to deliver, we select them and call the delivery primitive. Ready means they are in state $S3$, and the message either does not conflict with any other in the memory structure or is smaller than all others. Once a message is ready, we also collect the messages that do not conflict with any other for delivery in a single batch.

$$DoDeliver(g,\ p) \triangleq$$

We are accessing the buffer directly, and not through the *Memory instance.*
We do this because is easier and because we are only reading the values here.
Any changes we do through the instance.

$$\begin{array}{l}
\quad \exists\, \langle m_1,\ state,\ ts_1\rangle \in MemoryBuffer[g][p] :\\
\qquad \land\ state = \text{“S3”}\\
\qquad \land\ \forall\, \langle m_2,\ ignore,\ ts_2\rangle \in (MemoryBuffer[g][p] \setminus \{\langle m_1,\ state,\ ts_1\rangle\}) :\\
\qquad\quad \land\ \lor\ \neg CONFLICTR(m_1,\ m_2)\\
\qquad\qquad \lor\ ts_1 < ts_2 \lor (m_1.id < m_2.id \land ts_1 = ts_2)\\
\qquad \land\ \text{LET}\\
\qquad\quad G \triangleq Memory!ForAllFilter(g,\ p,\\
\qquad\qquad \text{LAMBDA}\ t_i,\ t_j : t_i[2] = \text{“S3”} \land \neg CONFLICTR(t_i[1],\ t_j[1]))\\
\qquad\quad D \triangleq G \cup \{\langle m_1,\ \text{“S3”},\ ts_1\rangle\}\\
\qquad\quad F \triangleq \{t[1] : t \in D\}\\
\qquad\quad \text{IN}
\end{array}$$

$\land$ $Memory!Remove(g, p, D)$
$\land$ $Delivered' = [Delivered \text{ EXCEPT } ![g][p] =$
$\quad Delivered[g][p] \cup Enumerate(Cardinality(Delivered[g][p]), F)]$
$\land$ UNCHANGED $\langle QuasiReliableChannel,$
$\quad GenericBroadcastBuffer, Votes, PreviousMsgs, K\rangle$

LOCAL $InitProtocol \triangleq$
$\land$ $K = [i \in Groups \mapsto [p \in Processes \mapsto 0]]$
$\land$ $Memory!Init$
$\land$ $PreviousMsgs = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$
$\land$ $Delivered = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$
$\land$ $Votes = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$

LOCAL $InitCommunication \triangleq$
$\land$ $GenericBroadcast!Init$
$\land$ $QuasiReliable!Init$

$Init \triangleq InitProtocol \land InitCommunication$

$Step(g, p) \triangleq$
$\lor$ $ComputeGroupSeqNumber(g, p)$
$\lor$ $GatherGroupsTimestamp(g, p)$
$\lor$ $DoDeliver(g, p)$

$GroupStep(g) \triangleq$
$\exists p \in Processes : Step(g, p)$

$Next \triangleq$
$\lor$ $\exists g \in Groups : GroupStep(g)$
$\lor$ UNCHANGED $vars$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$SpecFair \triangleq Spec \land \mathrm{WF}_{vars}(\exists g \in Groups : GroupStep(g))$

Helper functions to aid when checking the algorithm properties.

$WasDelivered(g, p, m) \triangleq$

Verifies if the given process p in group g delivered message m .

$\land$ $\exists \langle idx, n\rangle \in Delivered[g][p] : n.id = m.id$

$FilterDeliveredMessages(g, p, m) \triangleq$

Retrieve the set of messages with the same id as message m delivered by the given process p in group g .

$\{\langle idx,\, n\rangle \in Delivered[g][p] : n.id = m.id\}$

$DeliveredInstant(g,\, p,\, m) \triangleq$

Retrieve the instant the process p in group g delivered message m .

$(\textsc{choose}\ \langle t,\, n\rangle \in Delivered[g][p] : n.id = m.id)[1]$

A.6 TLC Executions

This section contains information about the models we checked using TLC. First, we show the TLA$^+$ specification we created for each algorithm's property. Then, we display all the information regarding the executions. Some checkings never finished executing. The conflict relations we used were *NeverConflict*, *AlwaysConflict*, and *IdConflict*.

A.6.1 Generic Multicast 0

Combinations with `NPROCESSES` and `NMESSAGES` were simultaneously greater than 3 take too much time to complete.

Table 2 – Generic Multicast 0 *Agreement* configurations.

NPROCESSES	NMESSAGES	CONFLICTR
2	2	All
2	3	All
3	2	All

Table 3 – Generic Multicast 0 configurations for remaining properties.

NPROCESSES	NMESSAGES	CONFLICTR
2	2	All
2	3	All
3	2	All
4	2	All

MODULE *Agreement*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _)

Since this algorithm is for failure-free environments, the set of all processes is the same as the correct ones.

LOCAL $Processes \triangleq \{i : i \in 1 \, .. \, NPROCESSES\}$
LOCAL $ChooseProcess \triangleq$ CHOOSE $x \in Processes$: TRUE
LOCAL $Create(id) \triangleq [id \mapsto id, \; d \mapsto Processes, \; o \mapsto ChooseProcess]$
LOCAL $AllMessages \triangleq \{Create(id) : id \in 1 \, .. \, NMESSAGES\}$

VARIABLES
    K,
    Pending,
    Delivering,
    Delivered,
    PreviousMsgs,
    Votes,
    QuasiReliableChannel

Initialize the instance for the Generic Multicast 0. The *INITIAL_MESSAGES* is a set with *NMESSAGES*, unordered, a tuple with the starting state $S0$ and the message.

$Algorithm \triangleq$ INSTANCE *GenericMulticast0* WITH
    $INITIAL\_MESSAGES \leftarrow \{\langle \text{"S0"}, \; m \rangle : m \in AllMessages\}$

Weak fairness is necessary.
$Spec \triangleq Algorithm!SpecFair$

If a correct process deliver a message m , then all correct processes in $m.d$ eventually delivers m .

We verify that all messages in *AllMessages*, for all the processes that delivered a message, eventually, all the correct members in the destination will deliver.

$$
\begin{aligned}
& Agreement \triangleq \\
& \quad \forall m \in AllMessages : \\
& \quad\quad \forall p \in Processes : \\
& \quad\quad\quad Algorithm!WasDelivered(p, \; m) \\
& \quad\quad\quad\quad \leadsto \forall q \in \{x \in m.d : x \in Processes\} : \\
& \quad\quad\quad\quad\quad Algorithm!WasDelivered(q, \; m)
\end{aligned}
$$

MODULE *Collision*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _)

Since this algorithm is for failure-free environments, the set of all processes is the same as the correct ones.

LOCAL *Processes* $\triangleq$ $\{i : i \in 1 \,..\, NPROCESSES\}$
LOCAL *ChooseProcess* $\triangleq$ CHOOSE $x \in Processes$: TRUE
LOCAL *Create*(*id*) $\triangleq$ $[id \mapsto id,\ d \mapsto Processes,\ o \mapsto ChooseProcess]$
LOCAL *AllMessages* $\triangleq$ $\{Create(id) : id \in 1 \,..\, NMESSAGES\}$

VARIABLES
    K,
    Pending,
    Delivering,
    Delivered,
    PreviousMsgs,
    Votes,
    QuasiReliableChannel

Initialize the instance for the Generic Multicast 0. The *INITIAL_MESSAGES* is a set with *NMESSAGES*, unordered, a tuple with the starting state $S0$ and the message.

Algorithm $\triangleq$ INSTANCE *GenericMulticast0* WITH
    $INITIAL\_MESSAGES \leftarrow \{\langle \text{"S0"}, m\rangle : m \in AllMessages\}$

Spec $\triangleq$ *Algorithm*!*Spec*

If a correct process p delivers messages m and n, p is in the destination of both messages, m and n do not commute. Then, p delivers either m and then n or n and then m.

Collision $\triangleq$
    $\Box \forall\, p \in Processes$:
        $\forall\, m, n \in AllMessages : \land\ m.id \neq n.id$
            $\land$ *Algorithm*!*WasDelivered*(p, m)
            $\land$ *Algorithm*!*WasDelivered*(p, n)
            $\land$ *CONFLICTR*(m, n)
                $\Rightarrow$ *Algorithm*!*DeliveredInstant*(p, m) $\neq$
                    Algorithm!*DeliveredInstant*(p, n)

```
------------------------------ MODULE Integrity ------------------------------
EXTENDS Naturals, FiniteSets, Commons

CONSTANT NPROCESSES
CONSTANT NMESSAGES
CONSTANT CONFLICTR(_, _)
------------------------------------------------------------------------------
```

Since this algorithm is for failure-free environments, the set of all processes is the same as the correct ones.

```
LOCAL Processes ≜ {i : i ∈ 1 .. NPROCESSES}
LOCAL ChooseProcess ≜ CHOOSE x ∈ Processes : TRUE
```

This property verifies that we only deliver sent messages. To assert this, we create $NMESSAGES+1$ and do not include the additional one in the algorithm execution, then check that the delivered ones are only the sent ones.

```
LOCAL AcceptableMessageIds ≜ {id : id ∈ 1 .. NMESSAGES}
LOCAL Create(id) ≜ [id ↦ id, d ↦ Processes, o ↦ ChooseProcess]
LOCAL AllMessages ≜ {Create(id) : id ∈ 1 .. (NMESSAGES + 1)}
LOCAL SentMessage ≜ {m ∈ AllMessages : m.id ∈ AcceptableMessageIds}
------------------------------------------------------------------------------
VARIABLES
    K,
    Pending,
    Delivering,
    Delivered,
    PreviousMsgs,
    Votes,
    QuasiReliableChannel
```

Initialize the instance for the Generic Multicast 0. The *INITIAL_MESSAGES* is a set with *NMESSAGES*, unordered, a tuple with the starting state $S0$ and the message.

```
Algorithm ≜ INSTANCE GenericMulticast0 WITH
    INITIAL_MESSAGES ← {⟨"S0", m⟩ : m ∈ SentMessage}
------------------------------------------------------------------------------
Spec ≜ Algorithm!Spec
------------------------------------------------------------------------------
LOCAL DeliveredOnlyOnce(p, m) ≜
    Cardinality(Algorithm!FilterDeliveredMessages(p, m)) = 1
```

For every message, all the correct processes in the destination deliver it only once, and a process previously sent it.

```
Integrity ≜
    □∀ m ∈ AllMessages :
        ∀ p ∈ Processes :
            Algorithm!WasDelivered(p, m) ⇒
                (DeliveredOnlyOnce(p, m) ∧ p ∈ m.d ∧ m ∈ SentMessage)
==============================================================================
```

MODULE *PartialOrder*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _)

Since this algorithm is for failure-free environments, the set of all processes is the same as the correct ones.

$$\text{LOCAL } Processes \triangleq \{i : i \in 1 \mathrel{..} NPROCESSES\}$$
$$\text{LOCAL } ChooseProcess \triangleq \text{CHOOSE } x \in Processes : \text{TRUE}$$
$$\text{LOCAL } Create(id) \triangleq [id \mapsto id,\ d \mapsto Processes,\ o \mapsto ChooseProcess]$$
$$\text{LOCAL } AllMessages \triangleq \{Create(id) : id \in 1 \mathrel{..} NMESSAGES\}$$

VARIABLES *K*, *Pending*, *Delivering*, *Delivered*,
PreviousMsgs, *Votes*, *QuasiReliableChannel*

Initialize the instance for the Generic Multicast 0. The *INITIAL_MESSAGES* is a set with *NMESSAGES*, unordered, a tuple with the starting state $S0$ and the message.

$$Algorithm \triangleq \text{INSTANCE } GenericMulticast0 \text{ WITH}$$
$$\quad INITIAL\_MESSAGES \leftarrow \{\langle \text{"S0"}, m\rangle : m \in AllMessages\}$$

$$Spec \triangleq Algorithm!Spec$$

$$\text{LOCAL } BothDelivered(p, q, m, n) \triangleq$$
$$\quad \land Algorithm!WasDelivered(p, m) \land Algorithm!WasDelivered(p, n)$$
$$\quad \land Algorithm!WasDelivered(q, m) \land Algorithm!WasDelivered(q, n)$$

$$\text{LOCAL } LHS(p, q, m, n) \triangleq$$
$$\quad \{p, q\} \subseteq (m.d \cap n.d) \land BothDelivered(p, q, m, n) \land CONFLICTR(m, n)$$

$$\text{LOCAL } RHS(p, q, m, n) \triangleq$$
$$\quad (Algorithm!DeliveredInstant(p, m) < Algorithm!DeliveredInstant(p, n))$$
$$\qquad \equiv (Algorithm!DeliveredInstant(q, m) < Algorithm!DeliveredInstant(q, n))$$

For every two messages, if they conflict, given a pair of processes, they are in the messages' destination, then both must deliver in the same order.

$$PartialOrder \triangleq$$
$$\quad \Box \forall p, q \in Processes :$$
$$\qquad \forall m, n \in AllMessages :$$
$$\qquad\quad LHS(p, q, m, n) \Rightarrow RHS(p, q, m, n)$$

MODULE *Validity*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NMESSAGES*
CONSTANT $CONFLICTR(\_, \_)$

Since this algorithm is for failure-free environments, the set of all processes is the same as the correct ones.

LOCAL $Processes \triangleq \{i : i \in 1 \,..\, NPROCESSES\}$
LOCAL $ChooseProcess \triangleq$ CHOOSE $x \in Processes$: TRUE
LOCAL $Create(id) \triangleq [id \mapsto id, d \mapsto Processes, o \mapsto ChooseProcess]$
LOCAL $AllMessages \triangleq \{Create(id) : id \in 1 \,..\, NMESSAGES\}$

VARIABLES
K,
Pending,
Delivering,
Delivered,
PreviousMsgs,
Votes,
QuasiReliableChannel

Initialize the instance for the Generic Multicast 0. The *INITIAL_MESSAGES* is a set with *NMESSAGES*, unordered, a tuple with the starting state $S0$ and the message.

$Algorithm \triangleq$ INSTANCE *GenericMulticast0* WITH
$INITIAL\_MESSAGES \leftarrow \{\langle \text{"S0"}, m\rangle : m \in AllMessages\}$

Weak fairness is necessary.

$Spec \triangleq Algorithm!SpecFair$

If a correct process GM-Cast a message m to $m.d$, then some process in $m.d$ eventually GM-Deliver m .

We verify that all messages on the messages that will be sent, then we verify that exists a process on the existent processes that did sent the message and eventually exists a process on $m.d$ *that delivers the message*.

$Validity \triangleq$
$\forall m \in AllMessages :$
$m.o \in Processes \leadsto \exists q \in m.d : Algorithm!WasDelivered(q, m)$

A.6.2 Generic Multicast 1

Table 4 – Generic Multicast 1 *Integrity* configurations.

NGROUPS	NPROCESSES	NMESSAGES	CONFLICTR
1	2	2	All
1	3	2	All
1	2	3	All
2	2	1	All

Table 5 – Generic Multicast 1 configurations for *Agreement* and *Validity*.

NGROUPS	NPROCESSES	NMESSAGES	CONFLICTR
1	2	2	All
1	3	2	All
1	2	3	All
1	2	4	All
2	2	1	All

Table 6 – Generic Multicast 1 configurations for *Partial Order* and *Collision*.

NGROUPS	NPROCESSES	NMESSAGES	CONFLICTR
1	2	2	All
1	3	2	All
1	2	3	All
1	2	4	All

MODULE *Agreement*

EXTENDS *Naturals*, *FiniteSets*, *Commons*, *TLC*

CONSTANT *NPROCESSES*
CONSTANT *NGROUPS*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _

This algorithm works in an environment with crash-stop failures, but we do not model processes failing. The set of all processes contains all correct ones.

LOCAL $Processes \triangleq \{i : i \in 1 \,..\, NPROCESSES\}$
LOCAL $Groups \triangleq 1 \,..\, NGROUPS$
LOCAL $ProcessesInGroup \triangleq [g \in Groups \mapsto Processes]$
LOCAL $AllMessages \triangleq CreateMessages(NMESSAGES, Groups, Processes)$
LOCAL $MessagesCombinations \triangleq CreatePossibleMessages(AllMessages)$

VARIABLES
K,
PreviousMsgs,
Delivered,
Votes,
MemoryBuffer,
QuasiReliableChannel,
AtomicBroadcastBuffer

Initialize the instance for the Generic Multicast 1. The *INITIAL_MESSAGES* is a sequence, totally ordered within a group, wherein the elements are tuples with the message, state, and timestamp.

$Algorithm \triangleq$ INSTANCE *GenericMulticast*1 WITH
$INITIAL\_MESSAGES \leftarrow [$
$g \in Groups \mapsto TotallyOrdered(MessagesCombinations[1])]$

$Spec \triangleq Algorithm!SpecFair$ Weak fairness is necessary.

If a correct process deliver a message m , then all correct processes in $m.d$ eventually delivers m .

We verify that all messages in *AllMessages*, for all the processes that delivered a message, eventually, all the correct members in the destination will deliver.

LOCAL $OnlyCorrects(g) \triangleq \{x \in ProcessesInGroup[g] : x \in Processes\}$

$$
\begin{aligned}
&Agreement \triangleq \\
&\quad \forall m \in AllMessages : \\
&\quad\quad \forall g\_i \in Groups : \\
&\quad\quad\quad \exists p\_i \in ProcessesInGroup[g\_i] : \\
&\quad\quad\quad\quad Algorithm!WasDelivered(g\_i, p\_i, m) \\
&\quad\quad\quad\quad\quad \leadsto \forall g\_j \in m.d : \exists p\_j \in OnlyCorrects(g\_j) : \\
&\quad\quad\quad\quad\quad\quad Algorithm!WasDelivered(g\_j, p\_j, m)
\end{aligned}
$$

```
------------------------------ MODULE Collision ------------------------------
EXTENDS Naturals, FiniteSets, Commons

CONSTANT NGROUPS
CONSTANT NPROCESSES
CONSTANT NMESSAGES
CONSTANT CONFLICTR(_, _)

-----------------------------------------------------------------------------

LOCAL Processes ≜ 1 .. NPROCESSES
LOCAL Groups ≜ 1 .. NGROUPS
LOCAL ProcessesInGroup ≜ [g ∈ Groups ↦ Processes]

LOCAL AllMessages ≜ CreateMessages(NMESSAGES, Groups, Processes)
LOCAL MessagesCombinations ≜ CreatePossibleMessages(AllMessages)

-----------------------------------------------------------------------------

VARIABLES K, PreviousMsgs, Delivered, Votes, MemoryBuffer,
    QuasiReliableChannel, AtomicBroadcastBuffer

(* Initialize the instance for the Generic Multicast 1. The INITIAL_MESSAGES is a sequence,
   totally ordered within a group, wherein the elements are tuples with the message, state, and
   timestamp. *)
Algorithm ≜ INSTANCE GenericMulticast1 WITH
    INITIAL_MESSAGES ← [
        g ∈ Groups ↦
            TotallyOrdered(MessagesCombinations[(g%NMESSAGES) + 1])]
-----------------------------------------------------------------------------

Spec ≜ Algorithm!Spec

-----------------------------------------------------------------------------
(* If a correct process p delivers messages m and n, p is in the destination of both messages, m and
   n do not commute. Then, p delivers either m and then n or n and then m. *)
Collision ≜
    □∀ g ∈ Groups :
        ∀ p ∈ ProcessesInGroup[g] :
            ∀ m1, m2 ∈ AllMessages : m1.id ≠ m2.id
                ∧ Algorithm!WasDelivered(g, p, m1)
                ∧ Algorithm!WasDelivered(g, p, m2)
                ∧ CONFLICTR(m1, m2)
                    ⇒ Algorithm!DeliveredInstant(g, p, m1) ≠
                        Algorithm!DeliveredInstant(g, p, m2)
=============================================================================
```

```
──────────────────────── MODULE Integrity ────────────────────────
EXTENDS Naturals, FiniteSets, Commons, Sequences

CONSTANT NPROCESSES
CONSTANT NGROUPS
CONSTANT NMESSAGES
CONSTANT CONFLICTR(_, _)

──────────────────────────────────────────────────────────────────
```

This algorithm works in an environment with crash-stop failures, but we do not model processes failing. The set of all processes contains all correct ones.

```
LOCAL Processes ≜ 1 .. NPROCESSES
LOCAL Groups ≜ 1 .. NGROUPS
LOCAL ProcessesInGroup ≜ [g ∈ Groups ↦ Processes]
```

This property verifies that we only deliver sent messages. To assert this, we create $NMESSAGES + 1$ and do not include the additional one in the algorithm execution, then check that the delivered ones are only the sent ones.

```
LOCAL AcceptableMessageIds ≜ {id : id ∈ 1 .. NMESSAGES}
LOCAL AllMessages ≜ CreateMessages(NMESSAGES + 1, Groups, Processes)
LOCAL SentMessage ≜ {m ∈ AllMessages : m.id ∈ AcceptableMessageIds}

LOCAL MessagesCombinations ≜ CreatePossibleMessages(AllMessages)
LOCAL CombinationsToSend ≜ [
    i ∈ DOMAIN MessagesCombinations ↦
      SelectSeq(MessagesCombinations[i], LAMBDA m : m ∈ SentMessage)]

──────────────────────────────────────────────────────────────────

VARIABLES
    K,
    PreviousMsgs,
    Delivered,
    Votes,
    MemoryBuffer,
    QuasiReliableChannel,
    AtomicBroadcastBuffer
```

Initialize the instance for the Generic Multicast 1. The *INITIAL_MESSAGES* is a sequence, totally ordered within a group, wherein the elements are tuples with the message, state, and timestamp.

```
Algorithm ≜ INSTANCE GenericMulticast1 WITH
    INITIAL_MESSAGES ← [g ∈ Groups ↦
        TotallyOrdered(CombinationsToSend[1])]
──────────────────────────────────────────────────────────────────

Spec ≜ Algorithm!Spec
```

LOCAL $DeliveredOnlyOnce(g, p, m) \triangleq$
$\quad Cardinality(Algorithm!FilterDeliveredMessages(g, p, m)) = 1$

For every message, all the correct processes in the destination deliver it only once, and a process previously sent it.

$Integrity \triangleq$
$\quad \Box \forall m \in AllMessages :$
$\qquad \forall g \in Groups :$
$\qquad\quad \forall p \in ProcessesInGroup[g] :$
$\qquad\qquad Algorithm!WasDelivered(g, p, m) \Rightarrow$
$\qquad\qquad\quad (DeliveredOnlyOnce(g, p, m) \land g \in m.d \land m \in SentMessage)$

```
---------------------------- MODULE PartialOrder ----------------------------
EXTENDS Naturals, FiniteSets, Commons

CONSTANT NPROCESSES, NGROUPS, NMESSAGES, CONFLICTR(_, _)
-----------------------------------------------------------------------------
LOCAL Processes ≜ 1 .. NPROCESSES
LOCAL Groups ≜ 1 .. NGROUPS
LOCAL ProcessesInGroup ≜ [g ∈ Groups ↦ Processes]

LOCAL AllMessages ≜ CreateMessages(NMESSAGES, Groups, Processes)
LOCAL MessagesCombinations ≜ CreatePossibleMessages(AllMessages)
-----------------------------------------------------------------------------
VARIABLES K, PreviousMsgs, Delivered, Votes, MemoryBuffer,
    QuasiReliableChannel, AtomicBroadcastBuffer

(* Initialize the instance for the Generic Multicast 1. The INITIAL_MESSAGES is a sequence,
   totally ordered within a group, wherein the elements are tuples with the message, state, and
   timestamp. *)
Algorithm ≜ INSTANCE GenericMulticast1 WITH
    INITIAL_MESSAGES ← [g ∈ Groups ↦
        TotallyOrdered(MessagesCombinations[(g%NMESSAGES) + 1])]
-----------------------------------------------------------------------------
Spec ≜ Algorithm!Spec
-----------------------------------------------------------------------------
LOCAL BothDelivered(g, p1, p2, m1, m2) ≜
    ∧ Algorithm!WasDelivered(g, p1, m1) ∧ Algorithm!WasDelivered(g, p1, m2)
    ∧ Algorithm!WasDelivered(g, p2, m1) ∧ Algorithm!WasDelivered(g, p2, m2)

LOCAL LHS(g, p1, p2, m1, m2) ≜
    ∧ {p1, p2} ⊆ (m1.d ∩ m2.d)
    ∧ CONFLICTR(m1, m2)
    ∧ BothDelivered(g, p1, p2, m1, m2)

LOCAL RHS(g, p1, p2, m1, m2) ≜
    (Algorithm!DeliveredInstant(g, p1, m1) <
        Algorithm!DeliveredInstant(g, p1, m2))
            ≡ (Algorithm!DeliveredInstant(g, p2, m1) <
                Algorithm!DeliveredInstant(g, p2, m2))

(* For every two messages, if they conflict, given a pair of processes, they are in the messages'
   destination, then both must deliver in the same order. *)
PartialOrder ≜
    □∀ g ∈ Groups :
        ∀ p1, p2 ∈ ProcessesInGroup[g] :
            ∀ m1, m2 ∈ AllMessages :
                LHS(g, p1, p2, m1, m2) ⇒ RHS(g, p1, p2, m1, m2)
=============================================================================
```

MODULE *Validity*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NGROUPS*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _)

LOCAL $Processes \triangleq 1 \,..\, NPROCESSES$
LOCAL $Groups \triangleq 1 \,..\, NGROUPS$
LOCAL $ProcessesInGroup \triangleq [g \in Groups \mapsto Processes]$

LOCAL $AllMessages \triangleq CreateMessages(NMESSAGES, Groups, Processes)$
LOCAL $MessagesCombinations \triangleq CreatePossibleMessages(AllMessages)$

VARIABLES *K*, *PreviousMsgs*, *Delivered*, *Votes*, *MemoryBuffer*, *QuasiReliableChannel*, *AtomicBroadcastBuffer*

Initialize the instance for the Generic Multicast 1. The *INITIAL_MESSAGES* is a sequence, totally ordered within a group, wherein the elements are tuples with the message, state, and timestamp.

$Algorithm \triangleq$ INSTANCE *GenericMulticast*1 WITH
$\quad INITIAL\_MESSAGES \leftarrow [$
$\qquad g \in Groups \mapsto$
$\qquad\quad TotallyOrdered(MessagesCombinations[(g \% NMESSAGES) + 1])]$

Weak fairness is necessary.

$Spec \triangleq Algorithm!SpecFair$

If a correct process GM-Cast a message m to $m.d$, then some process in $m.d$ eventually GM-Deliver m .

We verify that all messages on the messages that will be sent, then we verify that exists a process on the existent processes that did sent the message and eventually exists a process on $m.d$ *that delivers the message*.

$Validity \triangleq$
$\quad \forall m \in AllMessages :$
$\qquad m.o[1] \in Groups \land m.o[2] \in Processes$
$\qquad\quad \leadsto \exists g \in m.d :$
$\qquad\qquad \exists p \in ProcessesInGroup[g] : Algorithm!WasDelivered(g, p, m)$

A.6.3 Generic Multicast 2

Table 7 – Generic Multicast 1 *Integrity* configurations.

NGROUPS	NPROCESSES	NMESSAGES	CONFLICTR
1	2	2	All
1	3	2	All
1	2	3	All
2	2	1	All

Table 8 – Generic Multicast 1 configurations for *Agreement* and *Validity*.

NGROUPS	NPROCESSES	NMESSAGES	CONFLICTR
1	2	2	All
1	3	2	All
1	2	3	All
1	2	4	All
2	2	1	All

Table 9 – Generic Multicast 1 configurations for *Partial Order* and *Collision*.

NGROUPS	NPROCESSES	NMESSAGES	CONFLICTR
1	2	2	All
1	3	2	All
1	2	3	All
1	2	4	All

MODULE *Agreement*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NGROUPS*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _)

This algorithm works in an environment with crash-stop failures, but we do not model processes failing. The set of all processes contains all correct ones.

LOCAL $Processes \triangleq \{i : i \in 1 \,..\, NPROCESSES\}$
LOCAL $Groups \triangleq 1 \,..\, NGROUPS$
LOCAL $ProcessesInGroup \triangleq [g \in Groups \mapsto Processes]$

LOCAL $AllMessages \triangleq CreateMessages(NMESSAGES, Groups, Processes)$
LOCAL $MessagesCombinations \triangleq CreatePossibleMessages(AllMessages)$

VARIABLES
K,
PreviousMsgs,
Delivered,
Votes,
MemoryBuffer,
QuasiReliableChannel,
GenericBroadcastBuffer

Initialize the instance for the Generic Multicast 2. The *INITIAL_MESSAGES* is a sequence, partially ordered. The sequence elements are sets of messages, messages that commute can share a set.

$Algorithm \triangleq$ INSTANCE *GenericMulticast2* WITH
$INITIAL\_MESSAGES \leftarrow [g \in Groups \mapsto$
$PartiallyOrdered($
$MessagesCombinations[(g \% NMESSAGES) + 1], CONFLICTR)]$

Weak fairness is necessary.

$Spec \triangleq Algorithm!SpecFair$

If a correct process deliver a message m, then all correct processes in $m.d$ eventually delivers m.

We verify that all messages in *AllMessages*, for all the processes that delivered a message, eventually, all the correct members in the destination will deliver.

LOCAL $OnlyCorrects(g) \triangleq \{x \in ProcessesInGroup[g] : x \in Processes\}$
$Agreement \triangleq$
$\forall m \in AllMessages :$

$$
\begin{array}{l}
\forall\, g\_i \in Groups : \\
\quad \exists\, p\_i \in ProcessesInGroup[g\_i] : \\
\quad\quad Algorithm!WasDelivered(g\_i,\ p\_i,\ m) \\
\quad\quad\quad \leadsto \forall\, g\_j \in m.d : \\
\quad\quad\quad\quad \exists\, p\_j \in OnlyCorrects(g\_j) : \\
\quad\quad\quad\quad\quad Algorithm!WasDelivered(g\_j,\ p\_j,\ m)
\end{array}
$$

```
---------------------------- MODULE Collision ----------------------------
EXTENDS Naturals, FiniteSets, Commons

CONSTANT NGROUPS
CONSTANT NPROCESSES
CONSTANT NMESSAGES
CONSTANT CONFLICTR(_, _)

--------------------------------------------------------------------------
(* This algorithm works in an environment with crash-stop failures, but we do not model processes
   failing. The set of all processes contains all correct ones. *)
LOCAL Processes == 1 .. NPROCESSES
LOCAL Groups == 1 .. NGROUPS
LOCAL ProcessesInGroup == [g \in Groups |-> Processes]

LOCAL AllMessages == CreateMessages(NMESSAGES, Groups, Processes)
LOCAL MessagesCombinations == CreatePossibleMessages(AllMessages)

--------------------------------------------------------------------------

VARIABLES K, PreviousMsgs, Delivered, Votes, MemoryBuffer,
    QuasiReliableChannel, AtomicBroadcastBuffer

(* Initialize the instance for the Generic Multicast 2. The INITIAL_MESSAGES is a sequence,
   partially ordered. The sequence elements are sets of messages, messages that commute can share
   a set. *)
Algorithm == INSTANCE GenericMulticast2 WITH
    INITIAL_MESSAGES <- [g \in Groups |->
        PartiallyOrdered(
            MessagesCombinations[(g%NMESSAGES) + 1], CONFLICTR)]
--------------------------------------------------------------------------

Spec == Algorithm!Spec

--------------------------------------------------------------------------
(* If a correct process p delivers messages m and n, p is in the destination of both messages, m and
   n do not commute. Then, p delivers either m and then n or n and then m. *)
Collision ==
    []\A g \in Groups :
        \A p \in ProcessesInGroup[g] :
            \A m1, m2 \in AllMessages : m1.id # m2.id
                /\ Algorithm!WasDelivered(g, p, m1)
                /\ Algorithm!WasDelivered(g, p, m2)
                /\ CONFLICTR(m1, m2)
                    => Algorithm!DeliveredInstant(g, p, m1) #
                       Algorithm!DeliveredInstant(g, p, m2)
==========================================================================
```

MODULE *Integrity*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NGROUPS*
CONSTANT *NMESSAGES*
CONSTANT $CONFLICTR(\_, \_)$

This algorithm works in an environment with crash-stop failures, but we do not model processes failing. The set of all processes contains all correct ones.

LOCAL $Processes \triangleq 1 \,..\, NPROCESSES$
LOCAL $Groups \triangleq 1 \,..\, NGROUPS$
LOCAL $ProcessesInGroup \triangleq [g \in Groups \mapsto Processes]$

This property verifies that we only deliver sent messages. To assert this, we create $NMESSAGES + 1$ and do not include the additional one in the algorithm execution, then check that the delivered ones are only the sent ones.

LOCAL $AcceptableMessageIds \triangleq \{id : id \in 1 \,..\, NMESSAGES\}$
LOCAL $AllMessages \triangleq CreateMessages(NMESSAGES + 1, Groups, Processes)$
LOCAL $SentMessage \triangleq \{m \in AllMessages : m.id \in AcceptableMessageIds\}$

LOCAL $MessagesCombinations \triangleq CreatePossibleMessages(AllMessages)$
LOCAL $CombinationsToSend \triangleq [i \in \text{DOMAIN } MessagesCombinations \mapsto$
$SelectSeq(MessagesCombinations[i], \text{LAMBDA } m : m \in SentMessage)]$

VARIABLES
K,
PreviousMsgs,
Delivered,
Votes,
MemoryBuffer,
QuasiReliableChannel,
GenericBroadcastBuffer

Initialize the instance for the Generic Multicast 2. The *INITIAL_MESSAGES* is a sequence, partially ordered. The sequence elements are sets of messages, messages that commute can share a set.

$Algorithm \triangleq$ INSTANCE *GenericMulticast2* WITH
$INITIAL\_MESSAGES \leftarrow [g \in Groups \mapsto$
$PartiallyOrdered($
$CombinationsToSend[(g\%NMESSAGES) + 1], CONFLICTR)]$

$Spec \triangleq Algorithm!Spec$

For every message, all the correct processes in the destination deliver it only once, and a process previously sent it.

$$
\begin{array}{l}
\textsc{local}\ \mathit{DeliveredOnlyOnce}(g,\ p,\ m)\ \triangleq \\
\quad \mathit{Cardinality}(\mathit{Algorithm}!\mathit{FilterDeliveredMessages}(g,\ p,\ m)) = 1 \\
\mathit{Integrity}\ \triangleq \\
\quad \Box \forall\, m \in \mathit{AllMessages} : \\
\qquad \forall\, g \in \mathit{Groups} : \\
\qquad\quad \forall\, p \in \mathit{ProcessesInGroup}[g] : \\
\qquad\qquad \mathit{Algorithm}!\mathit{WasDelivered}(g,\ p,\ m) \Rightarrow \\
\qquad\qquad\quad (\mathit{DeliveredOnlyOnce}(g,\ p,\ m) \land g \in m.d \land m \in \mathit{SentMessage})
\end{array}
$$

MODULE *PartialOrder*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT $NGROUPS$, $NPROCESSES$, $NMESSAGES$, $CONFLICTR(\_, \_)$

This algorithm works in an environment with crash-stop failures, but we do not model processes failing. The set of all processes contains all correct ones.

$$
\begin{aligned}
&\text{LOCAL } Processes \triangleq 1 \,..\, NPROCESSES \\
&\text{LOCAL } Groups \triangleq 1 \,..\, NGROUPS \\
&\text{LOCAL } ProcessesInGroup \triangleq [g \in Groups \mapsto Processes] \\
&\text{LOCAL } AllMessages \triangleq CreateMessages(NMESSAGES, Groups, Processes) \\
&\text{LOCAL } MessagesCombinations \triangleq CreatePossibleMessages(AllMessages)
\end{aligned}
$$

VARIABLES K, $PreviousMsgs$, $Delivered$, $Votes$, $MemoryBuffer$, $QuasiReliableChannel$, $AtomicBroadcastBuffer$

Initialize the instance for the Generic Multicast 2. The *INITIAL_MESSAGES* is a sequence, partially ordered. The sequence elements are sets of messages, messages that commute can share a set.

$$
\begin{aligned}
&Algorithm \triangleq \text{INSTANCE } GenericMulticast2 \text{ WITH} \\
&\quad INITIAL\_MESSAGES \leftarrow [g \in Groups \mapsto \\
&\qquad PartiallyOrdered(\\
&\qquad\quad MessagesCombinations[(g \% NMESSAGES) + 1], CONFLICTR)]
\end{aligned}
$$

$$Spec \triangleq Algorithm!Spec$$

$$
\begin{aligned}
&\text{LOCAL } BothDelivered(g, p1, p2, m1, m2) \triangleq \\
&\quad \land Algorithm!WasDelivered(g, p1, m1) \land Algorithm!WasDelivered(g, p1, m2) \\
&\quad \land Algorithm!WasDelivered(g, p2, m1) \land Algorithm!WasDelivered(g, p2, m2) \\
&\text{LOCAL } LHS(g, p1, p2, m1, m2) \triangleq \\
&\quad \land \{p1, p2\} \subseteq (m1.d \cap m2.d) \\
&\quad \land CONFLICTR(m1, m2) \\
&\quad \land BothDelivered(g, p1, p2, m1, m2) \\
&\text{LOCAL } RHS(g, p1, p2, m1, m2) \triangleq \\
&\quad (Algorithm!DeliveredInstant(g, p1, m1) < \\
&\qquad Algorithm!DeliveredInstant(g, p1, m2)) \\
&\qquad\quad \equiv (Algorithm!DeliveredInstant(g, p2, m1) < \\
&\qquad\qquad Algorithm!DeliveredInstant(g, p2, m2))
\end{aligned}
$$

For every two messages, if they conflict, given a pair of processes, they are in the messages' destination, then both must deliver in the same order.

$$
\begin{aligned}
&PartialOrder \triangleq \\
&\quad \Box \forall\, g \in Groups : \\
&\qquad \forall\, p1, p2 \in ProcessesInGroup[g] : \\
&\qquad\quad \forall\, m1, m2 \in AllMessages : \\
&\qquad\qquad LHS(g, p1, p2, m1, m2) \Rightarrow RHS(g, p1, p2, m1, m2)
\end{aligned}
$$

MODULE *Validity*

EXTENDS *Naturals*, *FiniteSets*, *Commons*

CONSTANT *NPROCESSES*
CONSTANT *NGROUPS*
CONSTANT *NMESSAGES*
CONSTANT *CONFLICTR*(_, _)

This algorithm works in an environment with crash-stop failures, but we do not model processes failing. The set of all processes contains all correct ones.

$$\begin{array}{l}
\text{LOCAL } \mathit{Processes} \triangleq 1 \,..\, \mathit{NPROCESSES} \\
\text{LOCAL } \mathit{Groups} \triangleq 1 \,..\, \mathit{NGROUPS} \\
\text{LOCAL } \mathit{ProcessesInGroup} \triangleq [g \in \mathit{Groups} \mapsto \mathit{Processes}] \\
\\
\text{LOCAL } \mathit{AllMessages} \triangleq \mathit{CreateMessages}(\mathit{NMESSAGES}, \mathit{Groups}, \mathit{Processes}) \\
\text{LOCAL } \mathit{MessagesCombinations} \triangleq \mathit{CreatePossibleMessages}(\mathit{AllMessages})
\end{array}$$

VARIABLES *K*, *PreviousMsgs*, *Delivered*, *Votes*, *MemoryBuffer*,
QuasiReliableChannel, *AtomicBroadcastBuffer*

Initialize the instance for the Generic Multicast 2. The *INITIAL_MESSAGES* is a sequence, partially ordered. The sequence elements are sets of messages, messages that commute can share a set.

$$\begin{array}{l}
\mathit{Algorithm} \triangleq \text{INSTANCE } \mathit{GenericMulticast2} \text{ WITH} \\
\quad \mathit{INITIAL_MESSAGES} \leftarrow [g \in \mathit{Groups} \mapsto \\
\qquad \mathit{PartiallyOrdered}(\\
\qquad\quad \mathit{MessagesCombinations}[(g \% \mathit{NMESSAGES}) + 1],\ \mathit{CONFLICTR})]
\end{array}$$

Weak fairness is necessary.

$$\mathit{Spec} \triangleq \mathit{Algorithm}!\mathit{SpecFair}$$

If a correct process GM-Cast a message m to $m.d$, then some process in $m.d$ eventually GM-Deliver m .

We verify that all messages on the messages that will be sent, then we verify that exists a process on the existent processes that did sent the message and eventually exists a process on $m.d$ *that delivers the message.*

$$\begin{array}{l}
\mathit{Validity} \triangleq \\
\quad \forall\, m \in \mathit{AllMessages} : \\
\qquad m.o[1] \in \mathit{Groups} \land m.o[2] \in \mathit{Processes} \\
\qquad\quad \leadsto \exists\, g \in m.d : \\
\qquad\qquad \exists\, p \in \mathit{ProcessesInGroup}[g] : \mathit{Algorithm}!\mathit{WasDelivered}(g, p, m)
\end{array}$$