Agreement Abstractions in Anonymous and Homonymous Distributed Systems Prone to Failures

Doctoral Thesis

By

Jian Tang

Departamento de Sistemas Informáticos
Universidad Politécnica de Madrid

Co-supervised

By

Prof. Ph.D. Ernesto Jiménez and Prof. Ph.D. Sergio Arévalo

September 2015
Abstract

The distributed computing models typically assume every process in the system has a distinct identifier (ID) or each process is programmed differently, which is named as *eponymous system*. In such kind of distributed systems, the unique ID is helpful to solve problems: it can be incorporated into messages to make them trackable (i.e., to or from which process they are sent) to facilitate the message transmission; several problems (leader election, consensus, etc.) can be solved without the information of network property in priori if processes have unique IDs; messages in the register of one process will not be overwritten by others process if this process announces; it is useful to break the symmetry. Hence, eponymous systems have influenced the distributed computing community significantly either in theory or in practice. However, every thing in the world has its own two sides. The unique ID also has disadvantages: it can leak information of the network(size); processes in the system have no privacy; assign unique ID is costly in bulk-production(e.g, sensors). Hence, homonymous system is appeared. If some processes share the same ID and programmed identically is called *homonymous system*. Furthermore, if all processes shared the same ID or have no ID is named as *anonymous system*.

In homonymous or anonymous distributed systems, the symmetry problem (i.e., how to distinguish messages sent from which process) is the main obstacle in the design of algorithms. This thesis is aimed to propose different symmetry break methods (e.g., random function, counting technique, etc.) to solve *agreement* problem. *Agreement* is a fundamental problem in distributed computing including a family of abstractions. In this thesis, we mainly focus on the design of consensus, set agreement, broadcast algorithms in anonymous and homonymous distributed systems. Firstly, the fault-tolerant broadcast abstraction is studied in anonymous systems with reliable or fair lossy communication channels separately. Two classes of anonymous failure detectors $A\Theta$ and $A\Phi^*$ are proposed, and both of them together with a already proposed failure detector $\psi$ are implemented and used to enrich the system model to implement broadcast abstraction. Then, in the study of the consensus abstraction, it is proved the $A\Omega'$ failure detector class is strictly weaker than $A\Omega$ and $A\Omega'$ is implementable. The first implementation of consensus in anonymous asynchronous distributed systems augmented with $A\Omega'$ and where a majority of processes does not crash. Finally, a general
consensus problem– k-set agreement is researched and the weakest failure detector $L$ used to solve it, in asynchronous message passing systems where processes may crash and recover, with homonyms (i.e., processes may have equal identities), and without a complete initial knowledge of the membership.
# Table of contents

## List of figures

<table>
<thead>
<tr>
<th>List of figures</th>
<th>ix</th>
</tr>
</thead>
</table>

## List of tables

<table>
<thead>
<tr>
<th>List of tables</th>
<th>xi</th>
</tr>
</thead>
</table>

## 1 Introduction

1. Motivation .................................................. 3
   1.1 Agreement is Important in Distributed Systems ............... 4
   1.1.2 Agreement in Anonymous and Homonymous Distributed Systems 5
1.2 Symmetry in Anonymous and Homonymous Distributed Systems ..... 6
   1.2.1 A Short Story of Symmetry .................................. 6
   1.2.2 Symmetry Break in Anonymous and Homonymous Distributed Systems 7
1.3 About This Thesis ........................................... 8
   1.3.1 Aim of This Thesis ....................................... 8
   1.3.2 Contributions of This Thesis ............................. 9
   1.3.3 Organization of the Thesis ............................... 10

## 2 System Model and Definitions

2.1 Eponymous, Anonymous and Homonymous Distributed Systems ..... 13
2.2 Synchronous and Asynchronous System Model .................... 14
2.3 Process and Communication Models ............................. 16
2.4 Round-based Model ........................................ 16
2.5 Failure Models ............................................. 17
2.6 Coordination Abstractions .................................. 19
   2.6.1 Broadcast Abstractions .................................... 19
   2.6.2 Agreement Abstractions ................................. 21
2.7 Unreliable Failure Detectors ................................ 23
   2.7.1 Definition and Classification ........................... 23
   2.7.2 Failure Detectors Related to Agreement and Coordination Abstractions 24
# Table of contents

3 Fault-tolerant Broadcast in Anonymous Distributed Systems 27
  3.1 Introduction .............................................. 27
  3.2 System Model ............................................ 28
  3.3 Fault-tolerant Broadcast in Anonymous Distributed Systems with Reliable Communication Channels ........................................ 30
    3.3.1 The Algorithm of Reliable Broadcast in $AAS_{R_n, t}[\emptyset]$ ............... 32
    3.3.2 The Algorithm of URB in $AAS_{R_n, t}[t < n/2]$ .................................. 35
    3.3.3 Impossibility to Implement URB in $AAS_{R_n, t}[t > n/2]$ ...................... 39
    3.3.4 The Algorithm of URB in $AAS_{R_n, t}[\psi]$ .................................. 40
  3.4 Fault-tolerant Broadcast in Anonymous Distributed Systems with Fair Lossy Channels ........................................ 43
    3.4.1 The Algorithm of Reliable Broadcast in $AAS_{F_n, t}[\emptyset]$ ............... 43
    3.4.2 The Algorithm of Uniform Reliable Broadcast in $AAS_{F_n, t}[t < n/2]$ .... 46
    3.4.3 An Impossibility Result ...................................... 51
    3.4.4 Anonymous Failure Detector Class of $A\Theta$ .................................. 52
    3.4.5 The Algorithm of Uniform Reliable Broadcast in $AAS_{F_n, t}[A\Theta]$ ... 56
    3.4.6 Quiescent Fault-tolerant Broadcast Algorithms ................................ 59
  3.5 Related Work ............................................. 73

4 Consensus in Anonymous Distributed Systems 75
  4.1 Introduction .............................................. 75
  4.2 System Model ............................................ 75
  4.3 Failure Detector Class $A\Omega'$ ................................ 76
    4.3.1 Definition of $A\Omega'$ ...................................... 76
    4.3.2 $A\Omega'$ is Strictly Weaker than $A\Omega$ .................................. 77
    4.3.3 Anonymous Partially Synchronous System $APSS$ ............................... 79
    4.3.4 The Algorithm of $A\Omega'$ in $APSS$ ...................................... 79
    4.3.5 Correctness of $A_{A\Omega'}$ in $APSS$ .................................. 80
  4.4 Consensus with the Failure Detector $A\Omega'$ ................. 83
    4.4.1 Definition of Consensus ........................................ 83
    4.4.2 Anonymous Asynchronous System $AAS_{R_n, t}$ .............................. 84
    4.4.3 The Algorithm Consensus $A_{cons}$ in $AAS_{R_n, t}[A\Omega']$ ............. 84
    4.4.4 Correctness of the algorithm $A_{cons}$ in $AAS_{R_n, t}[A\Omega']$ ........... 86
    4.4.5 Analysis of rounds .............................................. 89
  4.5 Related Work ............................................. 90
5 Set Agreement in Homonymous Distributed Systems 93
  5.1 Introduction ........................................... 93
  5.2 System Model ......................................... 93
  5.3 Definitions ........................................... 96
  5.4 Implementing Set Agreement in the Crash-Recovery Model .... 97
    5.4.1 Description of the algorithm \( A_{\text{set}} \) ................. 97
    5.4.2 Proofs of \( A_{\text{set}} \) in \( \mathcal{H} \{ Z, \emptyset, n \} \) ........... 99
  5.5 On the Implementability of the Failure Detector \( \mathcal{L} \) in Crash-Recovery Model 102
  5.6 Implementing Failure Detector \( \mathcal{L} \) in the Crash-Recovery Model ........ 103
    5.6.1 System Model ..................................... 104
    5.6.2 Algorithm \( A_{\mathcal{L}} \) ................................ 104
  5.7 Related Work ......................................... 107

6 Conclusions and Future Work 109
  6.1 Summary of Results ..................................... 109
  6.2 Future Work ........................................... 110

References 113

Appendix A Anonymous Failure Detector Class \( A\Theta S \) and Its Application to Fault-tolerant Broadcast 123
  A.1 The Failure Detector Class \( A\Theta S \) .............................. 123
    A.1.1 The definition of the failure detector \( A\Theta S \) ................ 123
    A.1.2 The Algorithm of the Failure Detector \( A\Theta S \) .............. 124
  A.2 Fault-tolerant Broadcast in \( AAS_{F_n,t}[A\Theta S] \) ............... 125
    A.2.1 The Algorithm of RB in \( AAS_{F_n,t}[A\Theta S] \) ............... 125
    A.2.2 The Algorithm of URB in \( AAS_{F_n,t}[A\Theta S] \) .............. 127
List of figures

2.1 Eponymous, anonymous and homonymous distributed systems . . . . . . . 14
2.2 Failure Classification . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.1 The algorithm of RB in $AAS_{R_{n,t}}[\emptyset]$ (code of $p_i$) . . . . . . . 33
3.2 The algorithm of URB in $AAS_{R_{n,t}}[t < n/2]$ (code of $p_i$) . . . . . . . 37
3.3 The algorithm of URB in $AAS_{R_{n,t}}[\psi]$ (code of $p_i$) . . . . . . . 41
3.4 The algorithm of RB in $AAS_{F_{n,t}}[\emptyset]$ (code of $p_i$) . . . . . . . 45
3.5 The algorithm of URB in $AAS_{F_{n,t}}[\emptyset]$ (code of $p_i$) . . . . . . . 48
3.6 The algorithm of failure detector $A\Theta$ (code of $p_i$) . . . . . . . . 55
3.7 The algorithm of URB in $AAS_{F_{n,t}}[A\Theta]$ (code of $p_i$) . . . . . . . 58
3.8 The algorithm of failure detector $AP^*$ in $ASS_{R}[\emptyset]$ (code of $p_i$) . . . . . . . 62
3.9 Quiescent Algorithm of RB in $AAS_{F_{n,t}}[AP^*]$ (code for $p_i$) . . . . . . . 65
3.10 Quiescent algorithm of URB in $AAS_{F_{n,t}}[A\Theta, AP^*]$ (code of $p_i$) . . . . . . . 70

4.1 The algorithm of $A\Omega'$ in the system $APSS$ (code of $p_i$) . . . . . . . 80
4.2 The algorithm of consensus $A_{cons}$ in $AAS_{R_{n,t}}[\Omega']$ where is known that a majority of processes are correct (code of $p_i$) . . . . . . . 85

5.1 The algorithm of set agreement $A_{set}$ in $HAS_{f}[\mathcal{L}, \emptyset, n]$ . . . . . . . 98
5.2 The algorithm of failure detector $\mathcal{L}$ (code of $p_i$) . . . . . . . . 105

A.1 The algorithm of failure detector $A\Theta S$ in $AAS_{F_{n,t}}[\emptyset](code of p_i)$ . . . . . . . 125
A.2 Quiescent algorithm of RB in $AAS_{F_{n,t}}[A\Theta S]$ (code of $p_i$) . . . . . . . 126
A.3 Quiescent algorithm of URB in $AAS_{F_{n,t}}[A\Theta S]$ (code of $p_i$) . . . . . . . 128
# List of tables

1.1 Published papers .......................... 10
2.1 The eight classes of failure detectors .................. 24
Chapter 1

Introduction

Distributed (computing) systems is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages [57]. It develops very quickly in recent decades mainly because: (1) the size of processor is getting smaller but the performance is getting higher as the prediction of Moore’s law; (2) the appearance of a vast of computation intensive applications need more power of compute a traditional single processor can not afford; (3) the connection cost among processes is lower but faster than ever. Hence, the distributed systems is becoming more significant in our daily life, especially with the development of ubiquitous computing, big data and cloud computing.

In centralized computer systems, the failure of a processor will stop the service and cause the whole system fails down. On the contrary, the distributed systems can tolerant partial failures, i.e., the system can continue to work correctly with acceptability when a failure occurs. In order to build fault tolerant distributed systems, redundancy is used as the key technique [77] by replicating the components of the systems to a group of replicas located at different places. However, every coin has its two sides. Replication can cause inconsistent problem among replicas due to the communication delay or message lost, etc. This inconsistency can be solved by agreement protocols that replicas coordinate with each other, which makes the distributed systems works.

Informally, agreement protocols or problems (e.g., consensus [101], atomic commitment ([29], [114]), atomic broadcast ([80], [112]) and group membership [96] are fundamental in distributed systems, and have important significants in both theory and practice. Among them, consensus is the fundamental one because others distributed agreement problem can be reduced to it. Roughly, consensus means that each process proposes a value and all correct processes have to agree on a common value from one of the proposed values. This problem can be easily solved in synchronous distributed system with one crash failure because there
is a known bound on process computation time and communication delay. However, Fisher, Lynch and Paterson proved that it is impossible to solve deterministically in an asynchronous distributed systems (there is no time bound on process computation and communication delay), which is known as FLP impossibility [73]. In brief, this impossibility is due to the fact that it is impossible to distinguish a process is crashed or it runs very slow in asynchronous distributed systems. With this result, it seems that it is impossible to build a functional and reliable distributed system. Thanks to the hard work of researchers in distributed systems, a failure detector device is proposed by Chandra and Toueg [50] which is used to circumvent FLP impossibility result. Failure detector is a local device of each process that provides (may be unreliable) failure information of processes. Usually, the failure detector has its own mechanism to detect failures of processes.

As mentioned above, agreement problems have attracted a lot of well-deserved attention in the literature, and amount of them assume that all processes have distinct identifiers which is named as \textit{eponymous distributed systems}. In such kind of distributed systems, the unique ID is helpful to solve problems: it can be incorporated into messages to make them trackable (i.e., to or from which process they are sent) to facilitate the message transmission; several problems (leader election, consensus, etc.) can be solved without the information of network property in priori if processes have unique IDs; messages in the register of one process will not be overwritten by others process if this process announces; it is useful to break the symmetry. Hence, eponymous systems have influenced the distributed computing community significantly either in theory or in practice. However, the unique ID also has disadvantages: it can leak information of the network(size); processes in the system have no privacy; assign unique ID is costly in bulk-production(e.g, sensors). Therefore, what will happen if identifiers are not sufficient to assign uniquely to each process(\textit{homonymous or anonymous distributed systems}). In other words, how important an identifier in distributed systems. In fact, this identifier problem has been mentioned very early by D. Angluin [8] in 1980.

"How much does each processor in a network of processors need to know about its own identity, the identity of other processors and the underlying connection network in order for the network to be able to carry out useful functions?"

It is interesting to determine how identifiers are needed and useful to solve problems in distributed systems. After this seminal work, many results have been given in anonymous and homonymous distributed systems ([24], [37], [40], [41], [61]). Even though, comparing to the eponymous systems, anonymous and homonymous systems have not attracted sufficient attention in neither theoretical nor practical distributed systems. In contrast, the necessary of anonymous and homonymous distributed systems is very strong by now. The trend is caused
by the fast development of ubiquitous computing [9], the massive demand of privacy, for example, in some distributed applications as peer-to-peer file systems [54], users do not want to be identified [75], etc. Following this trend, the research in anonymous and homonymous distributed systems is very useful and valuable mainly due to [88]:

- The lesser identifiers necessary when designing the algorithms, the more environment the algorithm can be used (i.e., more adaptive).
- The lesser identifiers is used by the processes (components) of one system, the more distributed the system is.
- The lesser identifiers used in one system, the more privacy it has.

This thesis follows this develop trend and is devoted to the study of agreement abstractions and related problems in anonymous and homonymous distributed systems.

1.1 Motivation

Distributed computing mainly focus on what can be computed in a system composed of $n$ processes that can fail independently and normally are eponymous, i.e., they have unique identities. As mentioned above, distributed systems can be classified into two main kinds according to the number of different identifiers of processes: eponymous systems and homonymous systems. Eponymous system is one part of distributed systems, and the counterpart of it is homonymous system, i.e., some processes have no identifiers or no unique identifiers and all execute the same code. Additionally, the homonymous distributed system is called anonymous distributed system if all processes have no identifiers.

This thesis focus on how to implement agreement abstractions in anonymous and homonymous distributed systems and it is mainly motivated by two reasons: one is the importance of agreement problems but few studies of it in anonymous and homonymous distributed systems; another one is that we try to give a complete picture of distributed computing theory (in eponymous and homonymous distributed systems). More specially, it is well known that the design and implementation of fault-tolerant distributed systems is very challenging. Agreement protocol is an important block to build fault-tolerant systems, which has been well studied in eponymous distributed systems where each process has an unique identifier. This thesis not only extend the existed results in eponymous distributed system to homonymous distributed systems, but also find new results in homonymous distributed systems, such as impossibility results, lower bounds or the weakest failure detector of each agreement abstraction.
1.1.1 Agreement is Important in Distributed Systems

In distributed systems, processes are located in different places. In order to keep the whole system in consistent and fault-tolerant, all processes should reach a global agreement in their actions. It is well known that the most famous form of agreement problems is consensus, which refers that a set of distributed processes reach a common decision from their initial proposed values.

In order to solve agreement problems in distributed systems, two system models (or timing models) are usually considered: synchronous model and asynchronous model. Informally, a distributed system is synchronous if there is a time bound on message delay, clock drift, the time of a process to execute a step, and all these bounds are known by the processes; otherwise, it is an asynchronous system [50]. Agreement problem can be deterministically solved and also is relatively simple. Hence, in this thesis, we mainly focus on asynchronous systems model which is more common and more practical.

Consensus can not be solved deterministically in an asynchronous system even with a single crash process, which is named as FLP impossibility [73]. To circumvent this impossibility result, many works can be found in the literature where the asynchronous system is augmented with a failure detector [103] to achieve consensus. A failure detector [50] is a distributed tool that each process can invoke to obtain some information about process failures. There are many classes of failures detectors depending on the quality and type of the returned information.

The k-set agreement problem was initially proposed by [53], and it is considered as a generalization of consensus. This problem [53] guarantees that from \( n \) proposed values at most \( k \) can be decided by the processes. The \( k \)-set agreement problem that is trivial to solve when the maximum number of processes that may crash (denoted by \( t \)) is lesser than \( k \), or the maximum number of different proposed values (denoted by \( d \)) is equal or lesser than \( k \) (i.e., \( t < k \) or \( d \leq k \)), becomes impossible to solve in an asynchronous system where processes may crash when \( t \geq k \) and \( d > k \) ([42], [81], [110]). Like consensus, a failure detector is needed to circumvent this impossibility.

The nature character of distributed systems is "distributed", which brings one basic problem that how to communicate among processes. Many systems support a process to send a message to only one other process at one time. If this process \( p \) wants to send a message \( m \) to all processes in the system, it has to send \( m \) to each process separately. During this re-sending, inconsistency may arise if \( p \) crashes. Fault-tolerant broadcast abstraction can solve this kind of inconsistency problem, which is a fundamental service in distributed systems that helps to build reliable distributed applications. It is used to disseminate messages
1.1 Motivation

among a set of processes, and it has several forms according to their quality of service [46]. This abstraction accelerates the development of fault-tolerant applications.

1.1.2 Agreement in Anonymous and Homonymous Distributed Systems

The definitions of agreement problems are the same as in eponymous distributed systems. The difference is the implementation environment. In homonymous (anonymous) distributed systems, some (all) processes share the same identifier or do not have identifiers and execute the identical code. Hence, when a process receives a message, it can not distinguish this message comes from which sender. In message-passing distributed systems, the information of the sender and destination of a message is vital for dealing with any tasks. In order to solve the agreement problem in homonymous systems, the first issue needed to be solved is how to break the symmetry of the systems, i.e., how to distinguish messages from the same process or different processes.

By now, several works related to agreement problem have been done in anonymous and homonymous distributed systems. Many results have been given that related to leader election, byzantine consensus, failure detector, etc. ([61], [24], [37], [17], [41], [40]).

Some research has studied anonymous shared-memory systems when no failures can occur. Johnson and Schneider [85] gave leader election algorithms using versions of single-writer snapshots and test set objects. Attiya, Gorbach and Moran [23] gave a characterization of the tasks that are solvable without failures using registers if n is not known. Consensus is solvable in these models, but it is not solvable if the registers cannot be initialized by the programmer [82]. Aspnes, Fich and Ruppert [19] looked at failure-free models with other types of objects, such as counters. They also characterized which shared-memory models can be implemented if communication is through anonymous broadcasts, showing the broadcast model is equivalent to having shared counters and strictly stronger than shared registers.

There has also been some research on randomized algorithms for anonymous shared-memory systems with no failures. For the naming problem, processes must choose unique names for themselves. Processes can randomly choose names, which will be unique with high probability. Registers can be used to detect when the names chosen are indeed unique, thus guaranteeing correctness whenever the algorithm terminates, which happens with high probability ([91], [115]). Two papers gave randomized renaming algorithms that have finite expected running time, and hence terminate with probability 1 ([72], [89]).

Randomized algorithms for systems with crash failures have also been studied. Panconesi et al. [28] gave a randomized wait-free algorithm that solves the naming problem using single-writer registers, which give the system some ability to distinguish between different processes’ actions. Several impossibility results have been shown for randomized naming
using only multi-writer registers ([45], [72], [89]). Interestingly, Buhrman et al. [45] gave a randomized wait-free anonymous algorithm for consensus in this model that is based on Chandra's randomized consensus algorithm [48]. Thus, producing unique identifiers is strictly harder than consensus in the randomized setting. Aspnes, Shah and Shah [21] extended the algorithm of Buhrman et al. to a setting with infinitely many processes.

1.2 Symmetry in Anonymous and Homonymous Distributed Systems

Symmetry is always considered as a kind of beauty in architecture, music, painting or mathematics, etc. It has different definitions depending on the domain. Much of our understanding of the world is based on the perception and recognition of repeated patterns that are generalized by the mathematical concept of symmetries. However, in the field of designing distributed algorithms, symmetry is an obstacle that needed to be overcome.

In distributed systems, symmetry always mean that processes are the same, execute the identical determinist code without identifiers, or processes have the same identifier. In brief, there exists something (process or message) indistinguishable. According to this definition, eponymous distributed systems is asymmetry, and anonymous and homonymous distributed systems has some degree of symmetry depending on the number of identical identifiers. Hence, the anonymous distributed systems is a pure symmetry one.

Symmetry is a basic concept even in Theoretical Computer Science. For the design and analysis of the algorithms, the symmetry plays an important role. Sometimes, the incorporation of elegant methods of symmetry breaking into algorithms leads to the efficiency. Moreover, symmetric structures and patterns are omnipresent and the study of their algorithmic or computational aspects gives us a better understanding of the nature of the symmetry.

1.2.1 A Short Story of Symmetry

To break symmetry in distributed systems is like our human body. Our bodies start out symmetrical, the left side a perfect reflection of the right. At about six weeks, the embryo begins to be asymmetrical. The heart shows the first visible asymmetry. Starting out as a simple tube, it loops to the left. The heart then starts to grow different structures on each side, producing the chambers and vessels required to pump blood. Meanwhile, other organs start moving. The stomach and liver each move clockwise away from the midline of the embryo. The large intestines sprout an appendix on the right. The right lung grows three lobes, the
left only two. Actually, the motivation to find since when our body senses left from right can be traced back to 1788. Until recently, biologists have pinpointed that a single spot where the symmetry breaking of our body starts: a tiny pit called the node, on the embryo’s midline. The interior of the node is lined with hundreds of tiny hairs, called cilia, which whirl round and round at a rate of 10 times a second [119].

Until now, the secret of how and when our human body begins the asymmetry is still under research. Following this short story of symmetry and comparing to our human, the distributed systems is similar to embryo that the eponymous distributed systems is asymmetric by nature (with unique identifier) and homonymous especially anonymous distributed systems has an innate degree of symmetry. Hence, we try to find or create one “point” that as our body since when or where the symmetry breaking starts in this thesis. Based on this “point”, the distributed algorithms are designed in homonymous (anonymous) distributed systems.

### 1.2.2 Symmetry Break in Anonymous and Homonymous Distributed Systems

There are three main methods used to break symmetry in anonymous and homonymous distributed systems: randomization, leader election, and direction sensitive. Informally, randomization means that there is a random function subject to a distribution which is used to give random name to each process; leader election is a deterministic form of symmetry breaking that an elected leader can assign names to processes, count the number of processes of the system, etc.; direction sensitive refers to that each process has local port number, it senses the message received or sent from/to which port.

In the literature, the majority of researches assume that each process has a unique identifier (eponymous systems). Formally, it can be denoted by $n = l$, if this distributed systems is composed of $n$ processes and $l$ is the number of different identifiers. Hence, it is natural to ask one question: What problems can be solved in distributed computing if $1 \leq l \leq n$? In fact, leader election as the first problem was investigated firstly by D. Angluin [8] in 1980. Leader election is a basic method to break the symmetry of systems, and other problems (e.g., assign identifier to other processors, counting the number of processors in the network, etc.) can be solved easily once a leader is agreed by all correct processors. In this seminal paper, she used the graph theory to find what conditions are needed to establish a “center” if all processors are initially in the same state. By answering this question, the first impossibility result in anonymous distributed systems was given, i.e. it is impossible to elect a leader in a symmetric network. Then, Johnson and Schneider [86] introduced port labelings to processor (also is called as port awareness in [56]). Inspired by both results above, further research was
given by Yamashita and Kameda [118], in which more detailed characterization of solvable
problems was described. However, all these previous results are based on the assumption of
knowing the size of network a priori. Then, Boldi and Vigna [36] replaced this assumption
by arbitrary knowledge of the networks, and several extended results are given in ([35], [38]).

In [47], the termination of distributed algorithms in anonymous networks with arbitrary
knowledge is investigated. A self-stabilizing anonymous leader election algorithm in a tree is
proposed, depending on the behavior of a daemon. Moreover, leader election in anonymous
rings can be found in ([26], [68]).

Researches related to the anonymous and homonymous systems can be divided into
two main kinds according to the system model: shared memory and message passing. In
the message passing model, different problems are considered depending on the network
topology, e.g., ring, tree, complete graph.

1.3 About This Thesis

This thesis focus on how to design and implement the agreement abstractions in anonymous
and homonymous distributed systems, which is more general than eponymous system model.

1.3.1 Aim of This Thesis

There are four aims of this thesis:

The first aim is to explore theoretical results of distributed computing in anonymous and
homonymous distributed systems. More precisely, the representative distributed computing
problems, such as consensus, failure detector, and fault-tolerant broadcast, have been investig-
gated relatively well in eponymous systems model. Recently, anonymous and homonymous
distributed systems are developing very fast which is motivated by the demand of privacy and
the constraints of practice. Following this trend, it is meaningful and interesting to extend
the typical distributed computing problems from eponymous systems to anonymous and
homonymous systems.

The second aim is to understand not only the importance and advantages of unique ID,
but also the disadvantages of it.

The third aim is to find symmetry break methods for solving agreement abstractions
when unique ID is unavailable, i.e., in anonymous and homonymous distributed systems.

The fourth aim is to understand how to design and implement agreement abstractions
and the related failure detector devices in anonymous and homonymous distributed systems.
1.3 About This Thesis

1.3.2 Contributions of This Thesis

This thesis mainly has six contributions. For fault-tolerant broadcast, the fault-tolerant broadcast algorithms are designed firstly in anonymous distributed systems with reliable communication channels and an implementation of failure detector $\psi$ is also given; in the second part, we provide firstly two non-quiescent but simple algorithms in anonymous distributed systems, and then proposed two classes of anonymous failure detectors $A\Theta$ and $AP^*$, and two quiescent fault-tolerant broadcast algorithms are given finally. For consensus, a new algorithm of consensus with $A\Omega'$ is proposed in anonymous distributed systems with reliable communication channels. For K-set agreement, an algorithm of set agreement and an implementation of failure detector class of $L$ in homonymous distributed system with crash-recovery failure model are given.

1. An impossibility result related to uniform reliable broadcast.
   It is impossible to implement uniform reliable broadcast without a majority of correct processes in anonymous distributed systems, no matter with fair lossy or reliable communication channels. This impossibility result stems from the fact that it is impossible for a process to confirm a message has been received by at least one correct process before the uniform reliable broadcast of it. This is due to two reasons: asynchrony and anonymity.

2. Two new anonymous failure detector classes for fault-tolerant broadcast.
   We propose two new classes of anonymous failure detectors $A\Theta$, $AP^*$. $A\Theta$ is aimed to take place of the condition of a majority of correct processes of uniform reliable broadcast; $AP^*$ is designed to make the non-quiescent fault-tolerant broadcast algorithms to be quiescent. Furthermore, all the implementation algorithms of these failure detectors are also given.

3. New implementation algorithms of fault-tolerant broadcast abstraction in anonymous asynchronous distributed systems.
   The fault-tolerant broadcast algorithms are proposed in anonymous distributed systems with reliable communication channels firstly, and then in anonymous distributed systems with fair lossy communication channels.

4. An implementation algorithm of anonymous failure detector class of $A\Omega'$.
   We proved that failure detector $A\Omega'$ is strictly weaker than $A\Omega$. Then, it is implemented in anonymous partially synchronous distributed systems.

5. New implementation algorithm of consensus in anonymous asynchronous distributed systems.
This algorithm is designed in anonymous asynchronous distributed system that enriched with failure detector $A\Omega'$.

6. New implementation algorithm of set agreement in homonymous asynchronous distributed systems.

The first set agreement and failure detector $L$ algorithms are given in homonymous asynchronous distributed systems where process can crash/recovery and has incomplete knowledge of membership.

During the research period of this thesis, four journal papers have been published and four conference(workshop) papers. The details of these papers are shown in Table 1.1.

### 1.3.3 Organization of the Thesis

This thesis is organized by a series of chapters modified from journal/conference articles rooted in this research.

CHAPTER 1 is devoted to the motivation of this thesis by giving the background of agreement problems, anonymous and homonymous distributed systems, and the importance of symmetry break in anonymous and homonymous distributed systems.
CHAPTER 2 introduces the different system models of distributed computing and corresponding definitions, including synchrony, process, communication channels, failure models, failure detectors, etc.

CHAPTER 3 mainly focuses on the implementation of fault-tolerant broadcast abstraction in anonymous asynchronous distributed systems (AAS). The first part of this chapter focuses on the broadcast algorithms in AAS with reliable communication channels: (uniform) reliable broadcast abstraction algorithms running independently of the number of crashed processes are given; the second part of this chapter is devoted to the AAS with fair lossy communication channels: the non-quiescent implementation algorithms of reliable broadcast with any number of correct processes and uniform reliable broadcast with a majority of correct processes are given firstly; Then, for uniform reliable broadcast, a new class of failure detector $A \Theta$ is proposed in the case of minority of correct processes; Finally, anonymous failure detector $AP^*$ is proposed that can be used to implement quiescent fault-tolerant broadcast in AAS with fair lossy communication channels.

CHAPTER 4 presents that the $A \Omega'$ failure detector class is strictly weaker than $A \Omega$ (i.e., $A \Omega'$ provides less information about process crashes than $A \Omega$). We also present in this chapter the first implementation of $A \Omega'$ (hence, we also show that $A \Omega'$ is implementable), and, finally, we include the first implementation of consensus in AAS augmented with $A \Omega'$ and where a majority of processes does not crash.

CHAPTER 5 shows how to implement the set agreement abstraction. We have studied the set agreement problem, and the weakest failure detector $L'$ used to solve it, in asynchronous message passing systems where processes may crash and recover, with homonyms (i.e., processes may have equal identities), and without a complete initial knowledge of the membership.

CHAPTER 6 summarizes the results of this thesis and states the possible directions of the future work.
Chapter 2

System Model and Definitions

System model refers to a series of assumptions which is used to abstract the features, behaviors and constraints of a real system, and also, it is the foundation of description, analysis and design of any kind of systems. In order to ease the description and design work of this thesis, we first define the system model. Hence, this chapter is devoted to introduce the system model of distributed systems. Generally, the system models used by the theoretical study of distributed computing system can be categorized into two sets according to the mechanism of communication: message passing and shared memory. In message-passing model, processes communicate among each other by sending and receiving messages through communication channels; in the shared memory model, processes exchange messages by shared objects, such as registers, queues [52].

In this thesis, only the message-passing model is considered. We will present the common assumptions of this model in the literature and some previous research results related to this work.

2.1 Eponymous, Anonymous and Homonymous Distributed Systems

Normally, a distributed system is composed of \( n \) processes that messages passing through communication channels. Depending on the relationship between the number of processes and the number of identifiers, the distributed system can be classified into two classes. If all \( n \) processes have a unique identifier, it is named as eponymous distributed systems. A great quantity of researches in the literature uses such a system model. It is obvious that this kind of distributed system is innate asymmetric. Instead, if some of \( n \) processes share one identifier, the system model is named as homonymous distributed systems, which was
introduced in [64]. Specially, if all the processes have the same identifier (or have no identifier), it can also be named as anonymous system. The anonymous distributed system is a subset of homonymous distributed system. Different from eponymous distributed systems, homonymous distributed systems has some degree of symmetry by nature. The degree of symmetry depends on the number of processes sharing identical identifier, i.e., the more processes share one, the more symmetry the system has. The relationship among them can be found in Figure 2.1.

More formally, let \( ID \) be the set of different identities of all \( n \) processes. Then, \( 1 \leq |ID| \leq n \). So, in this system, \( id(i) \) can be equal to \( id(j) \) and \( p_i \) be different of \( p_j \) (we say in this cases that \( p_i \) and \( p_j \) are homonymous). Due to the fact that anonymous distributed system is a particular case of homonymy distributed system where all processes have the same identity, that is, \( id(i) = id(j) \), for all \( p_i \) and \( p_j \) of \( \Pi \) (i.e., \(|ID| = 1\)).

### 2.2 Synchronous and Asynchronous System Model

According to the time requirement of processes and communication channels, the distributed systems has two kinds of models: one has strong bounds on time which is called synchronous model; another one has no bounds on time which is called asynchronous model (or time-free model).

In brief, synchronous distributed systems encapsulates specific timing assumption. It refers that the time of message transit delay and process computation is bounded and known by all processes. This model is considered as an idealized model in which processes execute by round, i.e., each process sends messages, receives messages and does local computation in one round. In other words, a message is received in the very round in which it is sent [104]. More clear, it is defined as follows according to V. Hadzilacos and S. Toueg [80]:

![Fig. 2.1 Eponymous, anonymous and homonymous distributed systems](image-url)
2.2 Synchronous and Asynchronous System Model

- The time needed by each process to execute a step has a known lower and upper bounds.

- Each process has a local clock with known and bounded drift rate to the real time.

- There is a known bounded time delay for each message transmitted through a communication link.

Due to its synchrony, the agreement problems can be easily solved in this system model ([65], [87]).

Asynchronous Distributed Systems is defined with respect to the synchronous distributed system without any time assumption on one of the following three conditions:

- The execution time of a step of a process.

- The clock drift rate of each process.

- The message transmission delays through a communication link.

Moreover, the rest of distributed systems is neither synchronous nor asynchronous is called Partially Synchronous Distributed Systems. The first two kinds of such a system model were first introduced by Dwork et al. ([70], [71]) as follows:

1. Each process has a bound on the execution time of one step, the clock drift rate is bounded, and the message transmission delay is bounded, but all these bounds are not known.

2. Each process has a known bound on the execution time of one step, the clock drift rate is bounded and known, and the message transmission delay is bounded and known, but all these bounds hold only after an unknown time interval.

Inspired by the previous two models, the third model of partially synchronous system is proposed by T. D. Chandra and S. Toueg [50]:

3. Each process has a bound on the execution time of one step, the clock drift rate is bounded, and the message transmission delay is bounded, but all these bounds are not known and hold after an unknown time interval.
2.3 Process and Communication Models

The behavior of a computer program is modeled as a process [116]. A distributed system is consisted of a set of such processes, denoted by $\Pi = \{p_i\}_{i=1,...,n}$ such that its size $|\Pi|$ is $n$, and $i$ is the index of each process $p_i$, $1 \leq i \leq n$, each process running a particular algorithm.

A process is a state machine whose states are divided into variables, and that has a set of discrete actions which is classified as internal and external. External actions might be input actions (e.g., when a process receives a message) or output actions (e.g., when a process sends a message). All actions may change the state of a process. Each process is sequential in a sense that it deterministically run their algorithm instructions step by step.

Processes communicate with each other when running a distributed algorithm. There are two main communication models [116]: the message-passing model and the shared memory model.

**Message-passing model**: processes communicate with each other by sending and receiving messages.

**Shared memory model**: processes communicate by executing operation on shared objects (such as register or queues [52])

2.4 Round-based Model

In distributed systems, the process execute in synchronous or asynchronous rounds. We summarize the difference of them.

In synchronous system model, each of its executions consists of a sequence of rounds. There is a global variable $r$, provided by the model, that takes the successive integer values 1, 2, etc. A process can only read it. A round is made up of three consecutive phases [105]:

- A send phase in which each process sends messages. If, during a round $r$, a process crashes during that phase, only an arbitrary subset of the messages it was assumed to send can be received by their destination processes.

- A receive phase in which each process receives messages. The fundamental property of the synchronous model lies in the fact that a message sent by a process $p_i$ to a process $p_j$ at round $r$, is received by $p_j$ at the same round $r$.

- A computation phase during which each process processes the messages it received during that round and executes local computation.

The round-based model in asynchronous systems mainly has two differences [105] with respect to the synchronous model:
• As it is not given for free and implicitly managed by the model, the round variable $r$ has to be explicitly constructed and managed by the processes. To that end, each process $p_i$ handles a local variable $r_i$ that represents its local view of the current round number.

• The second difference concerns the receive phase. Due to the system asynchrony, there is no way for a process $p_i$ to know if a process $p_j$ has crashed or not (this information would allow $p_i$ not to be blocked forever when it waits for a message from $p_j$). So, each instance of the model has to fundamentally include a parameter (usually denoted $f$) specifying the maximum number of processes that can crash during a run. Given this additional information, during each round $r$, a process $p_i$ can wait for round $r$ messages from $(n - f)$ processes. This threshold ensures that $p_i$ can never be blocked forever.

### 2.5 Failure Models

In distributed systems, both process and communication channels may fail, i.e., the behavior of process or channel different from the predefined by the algorithm, then we call this process or channel is **faulty**; otherwise, it is **correct**. In this section, we will give the definition of failure models.

Each process in distributed systems mainly has four types of failures:

• Crash: a process stops to execute the algorithm any more. This failure model is also called as failure-stop, i.e., the process executes correctly before the point of crash; the process can not execute any code neither recover after the point of crash.

• Send omission: a process crashes or omits sending messages it was supposed to send.

• Receive omission: a process crashes or omits receiving messages it was supposed to receive.

• General omission: a process crashes or experience either send omission or receive omission.

• Byzantine: a process or communication channel exhibits arbitrary behavior that may send/transmit arbitrary messages at arbitrary times or commit omissions, or a process may stop or take an incorrect step.
Crash is one special case of omission failure, and byzantine failure includes all kinds of failures (as in Figure 2.2). A process is correct if it does not make any failure at all, i.e., it never crashes and experiences neither send nor receive omissions.

![Failure Classification](image)

Fig. 2.2 Failure Classification

Normally, a crashed process does not recovery in distributed systems. In some cases, a crashed process can recover, which is named as crash-recovery system model. This kind of model was firstly proposed in [3]. In this kind of model, a process is *down* while it is crashed, otherwise it is *up*. Let us define a *run* as the sequence of steps taken by processes while they are up. So, in every run, each process \( p_i \in \Pi \) belongs to one of these five classes:

- **Eventually-up**: process \( p_i \) crashes and recovers repeatedly a finite number of times (at least once), but eventually \( p_i \), after a recovery, never crashes again, remaining alive forever.

- **Permanently-up**: process \( p_i \) never crash and is alive forever.

- **Permanently-down**: process \( p_i \) is alive until it crashes, and it never recovers again.

- **Eventually-down**: process \( p_i \) crashes and recovers repeatedly for a finite number of times (at least once), but eventually \( p_i \), after a crash, never recovers again, remaining crashed forever.

- **Unstable**: process \( p_i \) crashes and recovers repeatedly an infinite number of times.

In a run, a permanently-down, eventually-down or unstable process is called as *incorrect*. On the other hand, a permanently-up or eventually-up process in a run is called as *correct*. The set of incorrect processes in a run is denoted by \( \text{Incorrect} \subseteq \Pi \), and the set of correct processes in a run is denoted by \( \text{Correct} \subseteq \Pi \). It is obvious that \( \text{Incorrect} \cup \text{Correct} = \Pi \).

For communication channels, they also have some failure models. In distributed systems, process communicate with each other by passing messages through communication channels. These channels can be classified according to their reliability as follows:
Reliable Channel: the channel between $p_i$ and $p_j$ is reliable if it satisfies the following three properties:

- Reliable delivery. Let $p_j$ be any process that sends a message $m$ to a process $p_j$. If neither $p_i$ nor $p_j$ crashes, then $p_j$ eventually delivers $m$.
- No duplication. No message is delivered by a process more than once.
- No creation. If a message $m$ is delivered by some process $p_j$, then $m$ was previously sent to $p_j$ by some process $p_i$.

Fair Lossy Channel: the channel between two processes $p$ and $q$ is called as fair lossy channel if it satisfies the following properties [5]:

- Fairness: If $p$ sends a message $m$ to $q$ an infinite number of times and $q$ is correct, then $q$ eventually receives $m$ from $p$.
- Uniform Integrity: If $q$ receives a message $m$ from $p$, then $p$ previously sent $m$ to $q$; and if $q$ receives $m$ infinitely often from $p$, then $p$ sends $m$ infinitely often to $q$.

### 2.6 Coordination Abstractions

#### 2.6.1 Broadcast Abstractions

Before introduce the broadcast abstractions, it is better to mention about group communication. It is an indirect communication paradigm guaranteeing all members of a group have to deliver a message when this message is sent to the group. The group membership of a group may changed and may be a subset of all processes in the system. Hence, it guarantees that a message is send to all the members of a group in one operation.

In contrast to group communication, broadcast is a service that a message is sent to all processes in the system(not the subset). The broadcast communication abstraction plays an important role in fault-tolerant distributed systems. It is used to disseminate messages among a set of processes, and it has several different forms according to its quality of service [46].

Best effort broadcast\footnote{Some authors considers that best effort does not guarantees any delivery property other than sending the messages. This means that anything can happen independently of senders failure. For example Ethernet and IP-cast hold this property.} with two communication operations, namely $send()$ and $receive()$, guarantees that all correct processes will deliver a message if and only if the sender is correct. That is to say, this abstraction does not offer any delivery guarantee if the sender
crashes, which may lead to an inconsistent view of the system state by different processes. To avoid this non-determinism in the delivery when the sender may crash, Reliable Broadcast (RB) with $RB\text{-broadcast()}$ and $RB\text{-deliver()}$ operations was introduced [113], offering some degree of delivery guarantee. In short, RB is a broadcast service that requires that all correct processes deliver the same set of messages, and that all messages sent by correct processes must be delivered by all correct processes. Note that RB only requires the correct processes to deliver the same set of messages, which still may cause inconsistency problems when a process $RB\text{-delivers}$ a message and then crashes. In order to avoid those inconsistencies, the strongest abstraction Uniform Reliable Broadcast (URB) was proposed by Hadzilacos and Toueg ([78], [79], [111]). Uniform Reliable Broadcast, with $URB\text{-broadcast()}$ and $URB\text{-deliver()}$ operations, guarantees that if a process (no matter correct or not) delivers a message $m$, then all correct processes deliver $m$.

In this thesis, we mainly focus on Reliable Broadcast and Uniform Reliable Broadcast. This two types of broadcast can be defined by their properties in the following way:

**Reliable Broadcast**  
Formally, reliable broadcast is defined by two primitives: $RB\_broadcast(m)$ and $RB\_deliver(m)$. They satisfy three properties as follows:

- **Validity**: If a correct process broadcasts a message $m$, then it eventually delivers $m$.

- **Agreement**: If a correct process delivers a message $m$, then all correct processes eventually deliver $m$.

- **Integrity**: For any message $m$, every correct process delivers $m$ at most once, and only if $m$ was previously broadcast by $sender(m)$.

**Uniform Reliable Broadcast**  
Uniform Reliable Broadcast offers complete delivery guarantees when spreading messages among processes. That is, when a process delivers a message $m$, then all correct processes have to deliver it. It is also defined in terms of two primitives: $URB\_broadcast(m)$ and $URB\_deliver(m)$. They satisfy the following three properties:

- **Validity**: If a correct process broadcasts a message $m$, then it eventually delivers $m$.

- **Uniform Agreement**: If some process delivers a message $m$, then all correct processes eventually deliver $m$.

- **Uniform Integrity**: For every message $m$, every process delivers $m$ at most once, and only if $m$ was previously broadcast by $sender(m)$.
2.6.2 Agreement Abstractions

Processes in distributed systems have to be coordinated with each other needs agreement. As mentioned by R. Michel [103], a simple agreement problem is called unique action problem that each process proposes an action to execute and they all have to execute once the very same action that has to be one of the actions has been proposed by one of them; another example is processes agree on one common deliver order of messages. In this section, we briefly summarize several forms of agreement abstractions.

• Consensus

Consensus is considered as a fundamental problem that needed to be solved no matter in theoretical or practical distributed systems fields when designing or implementing reliable application on top of distributed systems with unreliable behaviors [102]. It is defined originally by Lamport et al. in 1980 [337] in a round-based synchronous distributed system prone to byzantine failures assumes that all processes begin executing simultaneously with their initial values already in their states [22], i.e., every process proposes a value at the beginning and all non-faulty processes have to decide on one and the same value from the proposed values finally ([50], [73]). Formally, consensus can be defined as follows ([102], [90]):

- Termination: Every correct process eventually decides.
- Validity: If a process decides a value \( v \), then \( v \) was proposed by some process.
- Agreement: No two correct processes decide different values.

In the previous definition, the agreement property may cause a system to an inconsistent states because a fault process can decide a different value from correct processes before it fails. In order to solve such a disagreement problem, a strengthened form of agreement property is introduced which is named as uniform agreement property as follows:

- Uniform Agreement: No two processes (whether correct or correct) decide different values.

• K-set Agreement

The K-set Agreement problem is introduced firstly by Chaudhuri [53] that can be described as follows: in a distributed system composed of \( n \) processes and the maximum \( t \) of them can fail, each process proposes a value, and each non-faulty process has to decide a value from the proposed values, then no more than \( k \) different values are
decided. The consensus problem can be considered as 1-set agreement problem i.e., all non-faulty processes have to decide the same value [107]. More formally, It must fulfill the following properties:

- **Termination**: Every correct process eventually decides some value.
- **Validity**: If a process decides a value $v$, then $v$ was proposed by some process.
- **Agreement**: No more than $k$ different values are decided.

**Total Order Uniform Reliable Broadcast**

Total order uniform reliable broadcast (TO-URB) is the uniform reliable broadcast with additional delivery property that all processes deliver in the same order. It has two operations: $TO_{\text{broadcast}}()$ and $TO_{\text{deliver}}()$. The definition is given by the following properties:

- **Validity**: If a process TO-delivers a message $m$, then $m$ has been TO-broadcast by some process.
- **Integrity**: A process TO-delivers a message $m$ at most once.
- **Total order message delivery**: If a process TO-delivers a message $m$ and then TO-delivers a message $m'$, then no process TO-delivers $m'$ unless it has TO-delivers the message $m$.
- **Termination**: If a non-faulty process TO-broadcasts a message $m$ or a process TO-delivers a message $m$, then each non-faulty process TO-delivers the message $m$.

TO-URB connects the agreement problem with broadcast problem, and it is equivalent with consensus.

**Atomic Commitment**

Atomic commitment comes from the distributed database systems that all processes have to agree on whether to commit or abort transactions. Each process votes yes or no, then they make an agreed decision. It is defined by the following properties:

- **Agreement**: No two processes decide on different values.
- **Validity**:
  * If any process initially votes No, then Abort is the only possible decision.
  * If all processes vote Yes and there is no failure, then Commit is the only possible decision.
2.7 Unreliable Failure Detectors

The failure detector is a device that can provide some hints (unreliable) about which processes have crashed in distributed systems. The notion of failure detector is proposed and developed by Chandra and Toueg in their seminal paper [50]. Each process has access to its local failure detector for obtaining failure information of processes. A failure detector can make mistakes by wrongly suspect a running process as a crashed one or doesn’t suspect a really crashed process. Hence, the failure detector may repeatedly trust or suspect one process. This character of failure detector implies that any two failure detector of different processes may provide different failure information.

2.7.1 Definition and Classification

The failure detector is defined by two properties: completeness and accuracy. According to the two properties, it can be classified into eight classes that are shown in Table 2.1. Informally speaking, the completeness property means eventually a crashed process must be suspected by the failure detector, and accuracy property sets limitation on the mistakes it may make. A failure detector history \( H \) is a function from \( \Pi \times T \) to \( 2^\Pi \). \( H(p, t) \) is the value of the failure detector module of process \( p \) at time \( t \). If \( q \in H(p, t) \), it means that \( p \) suspects \( q \) at time \( t \) in \( H \). Totally, two completeness and four accuracy properties are given in [50]:

- Completeness:
  - Strong Completeness. Eventually every process that crashes is permanently suspected by every correct process.
  - Weak Completeness. Eventually every process that crashes is permanently suspected by some correct process.

- Accuracy:
  - Strong Accuracy. No process is suspected before it crashes.
  - Weak Accuracy. Some correct process is never suspected.
  - Eventually Strong Accuracy. There is a time after which correct processes are not suspected by any correct process.
Eventually Weak Accuracy. There is a time after which some correct process is never suspected by any correct process.

Each process has access to its local failure detector for obtaining failure information of processes. They can be divided into different classes according to the quality of information they give. A failure detector can make mistakes by wrongly suspect a running process as a crashed one or does not suspect a really crashed process. Hence, the failure detector may repeatedly trust or suspect one process. This character of failure detector implies that any two failure detector of different processes may provide different failure information.

<table>
<thead>
<tr>
<th>Completeness</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>Perfect $P$</td>
</tr>
<tr>
<td>Weak</td>
<td>Quasi Perfect $Q$</td>
</tr>
</tbody>
</table>

Table 2.1 The eight classes of failure detectors

2.7.2 Failure Detectors Related to Agreement and Coordination Abstractions

In this subsection, we summarize several important unreliable failure detectors proposed in the literature for agreement and coordination abstractions.

- **Failure Detectors for Consensus**

The weakest failure detector class to solve consensus in asynchronous message-passing distributed system prone to process crashes is $\Sigma \times \Omega$. This means the combination of class $\Sigma$ and $\Omega$. The failure detector class $\Sigma$ is the class of quorum failure detectors providing each process with a quorum that eventually each contains only non-faulty processes and any two quorums have at least one common process if they do intersect. The failure class $\Omega$ is the class of eventual leader failure detectors providing each process with a variable that eventually all these local variables contain the same set of non-faulty processes forever. The formal definition of these two classes of failure detector are as follows:

1. Failure detector class $\Sigma$:
   - Intersection. $\forall i, j \in \{1, ..., n\}: \forall \tau, \tau' \in \mathrm{IN}: \sigma_i^{\tau} \cap \sigma_j^{\tau'} \neq \emptyset.$
2.7 Unreliable Failure Detectors

- Liveness. \( \exists \tau \in \text{IN}: \forall \tau': \tau \geq \tau: \forall i \in \text{Correct}(F): \sigma_i^{\tau'} \subseteq \text{Correct}(F) \).

2. Failure detector class \( \Omega \):

- Validity. \( \forall i: \forall \tau: \text{leader}_i^\tau \) contains a process identity.
- Eventual leadership. \( \exists l \in \text{Correct}(F), \exists \tau: \forall \tau': \tau \geq \tau: \forall i \in \text{Correct}(F): \text{leader}_i^{\tau'} = l \).

Besides this, M. K. Aguilera, W. Chen, S. Toueg proposed failure detector \( \odot S_e \) and a stronger form \( \odot S_u \) for asynchronous distributed systems in which processes may crash and recover and links may lose messages [3]. As the authors mentioned that this failure detector is well-suited to the crash-recovery model and does not output lists of processes suspected to be crashed or unstable. Instead, it outputs a list of processes deemed to be currently up, with an associated epoch number for each such process. It satisfies two properties: (1) Completeness: For every bad process \( b \), at every good process there is a time after which either \( b \) is never trusted or the epoch number of \( b \) keeps on increasing; (2) Accuracy: Some good process is eventually trusted forever by all good processes, and its epoch number stops changing. The stronger \( \odot S_u \) has Strong Accuracy: Some good process is eventually trusted forever by all good and unstable processes, and its epoch number stops changing. Formally, it is defined as follows:

3. Failure detector class \( \odot S_e \) and \( \odot S_u \):

- Monotonicity. \( \forall F, \forall H \in D(F), \forall g \in \text{good}(F), \forall p \in \Pi, \exists T \in T, \exists t, t' > T: [p \in H(g, t).\text{trustlist} \land p \in H(g, t').\text{trustlist} \land t < t'] \Rightarrow H(g, t).\text{epoch}[p] \leq H(g, t').\text{epoch}[p] \).
- Completeness. \( \forall F, \forall H \in D(F), \forall b \in \text{bad}(F), \forall g \in \text{good}(F): [\exists T \in T, \forall t > T, b \notin H(g, t).\text{trustlist}] \lor [\forall M \in N, \exists t \in T, b \in H(g, t).\text{trustlist} \land H(g, t).\text{epoch}[b] > M] \).
- Accuracy. \( \forall F, \forall H \in D(F), \exists K \in \text{good}(F), \forall g \in \text{good}(F), \exists M \in N, \exists T \in T, \forall t > T: K \in H(g, t).\text{trustlist} \land H(g, t).\text{epoch}[K] = M \).
- Strong Accuracy. \( \forall F, \forall H \in D(F), \exists K \in \text{good}(F): [\forall p \in \text{good}(F), \exists M \in N, \exists T \in T, \forall t > T: K \in H(p, t).\text{trustlist} \land H(p, t).\text{epoch}[K] = M] \land [\forall u \in \text{unstable}(F), \exists M \in H(u, t).\text{trustlist} \land H(u, t).\text{epoch}[K] = M] \).

- Failure Detectors for Set Agreement

The weakest failure detector class \( L \) in message-passing distributed systems for set agreement is proposed in [66]. Failure detector \( L \) outputs, whenever queried by a process, one of two values: “true” or “false” such that the following two properties
are satisfied: (1) there is at least one process where the output is always “false”, and (2) if only one process is correct (does not crash), then the output at this process is eventually “true” forever.

4. Failure detector class $L$: The range of $L$ is $\{true, false\}$. For every environment $\varepsilon$, for every failure pattern $F \in \varepsilon$, and every history $H \in L(F)$:
   - $\exists p_i \in \Pi, \forall t, H(p_i, t) \neq true$.
   - $\forall p_i \in \Pi, correct(F) = \{p_i\} \Rightarrow \exists t, \forall t' \geq t, H(p_i, t') = true$.

• Failure Detectors for Broadcast

In order to implement uniform reliable broadcast abstraction without the assumption of a majority of correct processes in asynchronous distributed systems that processes are prone to crash and the communication channels are fair lossy, a failure detector class $\Theta$ was proposed by M.K. Aguilera, S. Toueg and B. Deianov [5]. This class of failure detector provides each process with a read-only local variable $trusted$ that always include one non-faulty process and eventually all variables only contain non-faulty processes. The formal definition is as follows:

5. Failure detector class $\Theta$:

   - Accuracy. $\forall i \in \Pi; \forall \tau \in IN: (trusted^\tau_i \cap Correct(F) \neq \emptyset)$.
   - Liveness. $\exists \tau \in IN: \forall \tau' \geq \tau: \forall i \in Correct(F): trusted^\tau_i \subseteq Correct(F)$.

Another class failure detector for broadcast is HB of heartbeat failure detectors [2] that is used to obtain quiescent broadcast abstraction. It is defined by the following two properties:

6. Failure detector class $HB$:

   - Completeness. $\forall i, j \in \Pi; \forall \tau \in IN: HB^\tau_i[j] \leq HB^{\tau+1}_i[j]$.
   - Liveness.
     * $\forall i, j \in \Pi; \forall \tau \in IN: HB^\tau_i[j] \leq HB^{\tau+1}_i[j]$.
     * $\forall i, j \in Correct(F); \forall K: \exists \tau \in IN: HB^{\tau+1}_i[j] > K$.

With these two classes of failure detector, the uniformity and quiescence properties of broadcast can be obtained in asynchronous distributed systems that processes are prone to crash and communication channels are fair lossy: the failure class $\Theta$ is used to guarantee that a message is delivered by a process will be delivered by all correct processes; the failure class $HB$ is used to guarantee all correct have delivered the same message and terminate the repeated broadcast.
Chapter 3

Fault-tolerant Broadcast in Anonymous Distributed Systems

3.1 Introduction

Fault-tolerant Broadcast is a basic service in distributed systems that helps to build reliable distributed applications. It is used to disseminate messages among a set of processes, and it has several forms according to their quality of service [46]. The weakest form is Reliable Broadcast (RB) with RB-broadcast() and RB-deliver() operations. It was introduced in [113], which offers some degree of delivery guarantees. In short, RB is a broadcast service that requires that all non-crashed processes deliver the same set of messages, and that all messages sent by non-crashed processes must be delivered by all non-crashed processes. That situation can cause some inconsistencies when a process crashes after RB-deliver() a message. For example, if a process RB-deliver(m), and it crashes after that, it is possible that none non-crashed process never executes RB-deliver(m). To tackle it, there is a stronger abstraction called Uniform Reliable Broadcast that was proposed by V. Hadzilacos and S. Toueg ([78], [79], [111]), with URB-broadcast() and URB-deliver() operations. It guarantees that if a process (no matter crashed or non-crashed) delivers a message $m$, then all non-crashed processes must deliver this message $m$.

Fault-tolerant broadcast service has been extensively studied in non-anonymous system, where each process has a unique identifier. However, the development of anonymous systems is very quick in the field of distributed systems. There is great demand and significant to research it in anonymous distributed systems. In general, the appearance and development of anonymous distributed systems are caused by two important reasons: privacy and practical constraints. In some distributed applications, like peer-to-peer file systems, users do not want
to be identified [75]. Other applications that use sensor networks has constraints in where a unique identity is not possible to be embedded in each sensor node (due, for example, to small storage capacity, reduced computational capacity, or a huge number of elements to be identified) [9]. As we have known, the first paper studied about anonymous systems was addressed by D. Angluin [8]. Then, lots of paper appeared in this field, as ring anonymous networks, and shared memory anonymous systems ([45], [67], [76], [60]).

In classic message passing distributed systems, processes communicate with each other by sending messages. Because they all have unique identifiers, senders can choose the recipients of their messages, and recipients are aware of the identities of the senders of messages they receive [10]. However, all these early rules have to be changed in anonymous systems. In this chapter, the anonymous message passing systems is considered as a system that is composed of a set of processes that can crash, where processes have no identifiers, and each process executes the same code.

In this chapter, we firstly study fault-tolerant broadcast service in anonymous distributed systems with reliable communication channels (if a process $p$ sends a message to process $q$, and $q$ is correct, then $q$ eventually receives $m$) or quasi-reliable (if a process $p$ sends a message to process $q$, and the two processes are correct, then $q$ eventually receives $m$) [111]. However, real channels are neither always reliable nor quasi-reliable, most of them are unreliable (e.g., fair lossy, which means that if a message of one type is sent an infinite number of times, then the channel will deliver this message for an infinite times [28]). Many works have been done to construct reliable channels over unreliable channels in non-anonymous systems ([28], [1]). Hence, in the second part of this chapter, the study of fault-tolerant broadcast service has been done in anonymous distributed systems with fair lossy communication channels.

Additionally, the anonymous distributed systems is a subset of homonymous distributed systems, i.e., all processes in homonymous systems are assigned the same ID is equivalent to anonymous systems. Therefore, if fault tolerant broadcast can be solved in anonymous distributed systems, then it also can be solved in homonymous distributed systems.

### 3.2 System Model

In this chapter, we consider a system model as follows: an anonymous asynchronous distributed system is composed of processes without identifiers and communicate with each other by passing messages via a completely connected network with reliable or fair lossy channels. Two primitives are used in this system to send and receive messages: $\text{broadcast}(m)$ and $\text{deliver}(m)$. We say that a process $p_i$ broadcasts a message $m$ when
it invokes $broadcast_i(m)$. Similarly, a process $p_i$ delivers a message $m$ when it invokes $deliver_i(m)$.

It is necessary to explain the difference between the two primitives receive and deliver in distributed computing. Normally, there is a protocol layer between the network and application which is used to give some guarantees, e.g., uniform, causal order to $Broadcast$. This protocol layer might delay messages that have arrived from the network, omit messages, use additional control messages, etc.. Hence, we call receive (and send) when messages is received (and output) by the protocol layer from (to) the network; deliver (and broadcast) when a message is received (and send) by the application from (to) the protocol layer. Briefly, in order to distinguish the messages arrival at different layers, we say receive means the message is received in protocol layer and deliver means the message is received at the application layer.

**Process** In this chapter, the anonymous distributed system is formed by a set of $n$ anonymous processes, denoted as $\Pi = \{p_i\}_{i=1,...,n}$, such that its size is $|\Pi| = n$. We denote $t$ as the maximum number of processes that can crash. For subsection 3.4.2, we are going to consider that a majority of processes never crashes in the system (i.e., $t > n/2$). We consider that $i$ ($1 \leq i \leq n$) is the index of each process of the system. All processes are anonymous, that means they have no identifiers and execute the same algorithm. The index $i$ of process cannot be known by any process of the system. We just use it as a notation like $p_1, \cdots, p_n$ to simplify the description of the algorithms. Furthermore, all processes are asynchronous, that is, there is no assumption on their respective speeds.

There is a global clock whose values are the positive natural numbers, which is used for notation, processes cannot check or modify it.

**Failure Model** A process that does not crash in a run is correct in that run, otherwise it is faulty. We use Correct to denote the set of correct processes in a run, and Faulty to denote the set of faulty processes. A process executes its algorithm correctly until it crashes. A crashed process can neither execute any more statements nor recover.

**Communication** Each pair of processes are connected by bidirectional reliable or fair lossy channels. Processes communicate among them by sending and delivering messages through these channels. Furthermore, in anonymous distributed systems, when a process delivers a message, it cannot determine who is the sender of this message.

**Reliable Channel** Reliable channel between $p_i$ and $p_j$ satisfies three properties:

- Reliable delivery. Let $p_j$ be any process that sends a message $m$ to a process $p_j$. If neither $p_i$ nor $p_j$ crashes, then $p_j$ eventually delivers $m$.
- No duplication. No message is delivered by a process more than once.
• No creation. If a message \( m \) is delivered by some process \( p_j \), then \( m \) was previously sent to \( p_j \) by some process \( p_i \).

**Fair Lossy Channel** A channel between two processes \( p \) and \( q \) is called as fair lossy channel if it satisfies the following properties [6]:

- Fairness: If \( p \) sends a message \( m \) to \( q \) an infinite number of times and \( q \) is correct, then \( q \) eventually delivers \( m \) from \( p \).

- Uniform Integrity: If \( q \) receives a message \( m \) from \( p \), then \( p \) previously sent \( m \) to \( q \); and if \( q \) delivers \( m \) infinitely often from \( p \), then \( p \) sends \( m \) infinitely often to \( q \).

**Anonymous message and Identical Message** A message is usually composed by two parts: the header and the value. But in anonymous system, all messages may have the same header or have no header. We define that all messages transmitted in anonymous systems are called anonymous messages. The definition of identical anonymous message is as follows:

If two messages \( m_1 \) and \( m_2 \) are composed with the same value, then we call them as identical message.

The most challenging work in anonymous distributed system is how to distinguish anonymous messages and how to count the identical anonymous messages.

**Random Function** In the subsection 3.4, a random function is managed by each process which can generate a global unique ID to each message it broadcast. In this thesis, we just consider this random function exists, and do not focus on how to implement such a function.

**Notation** The system model is denoted by \( \text{AAS}_R[n/0] / \text{AAS}_F[n/0] / \text{AAS}_F[n][D] \). \( \text{AAS}_R \) is an acronym for anonymous asynchronous message passing distributed systems with reliable communication channels; \( \text{AAS}_F \) is an acronym for anonymous asynchronous message passing distributed systems with fair lossy communication channels; \( \emptyset \) means there is no additional assumption, \( D \) means the system is enriched with a failure detector class of D. The variable \( n \) represents the total number of processes in the system, and \( t \) represents the maximum number of processes that can crash.

### 3.3 Fault-tolerant Broadcast in Anonymous Distributed Systems with Reliable Communication Channels

In this section, fault-tolerant broadcast services are studied in anonymous distributed systems with reliable communication channels. We first specify the system model. The anonymous
asynchronous system (denoted AAS[0]) is formed by a set of processes Π={pi}i=1,...,n such that its size |Π| is n, and i is the index of each process pi, 1 ≤ i ≤ n.

Processes are anonymous [41]. Hence, they have no identity, and there is no a way to differentiate between any two processes of the system (i.e., processes have no identifier, and execute the same code). So, anonymity implies that process indexes are fictitious in the sense that each process pi ∈ Π does not know its index i. We only use process indexes from an external observer point of view, and with the purpose of simplifying the notation.

A run R is formed by the set of steps taken by each process pi ∈ Π. We assume that time advances at discrete steps in each run R, and there is a global clock T whose values are the positive natural numbers. Note that T is an auxiliary concept that we only use for notation, but that processes can not check or modify. Processes are asynchronous, that is, the time to execute a step by a process in a run R is unbounded.

When a process crashes it stops taking steps. We assume that a crashed process never recovers. A process pi ∈ Π is correct if it does not crash, and faulty if it crashes. Let Correct be the set of correct processes, and let Faulty be the set of faulty processes. We denote by f the maximum number of processes that may crash. Unless otherwise is stated, we consider that this maximum number is n−1 (i.e., f ≤ n−1).

In AAS[0], processes communicate among them sending and receiving messages through reliable channels. The system AAS[0] has two primitives to send and receive messages: bcast(m) and del(m). We say that a process pi broadcasts a message m when it invokes bcasti(m). Similarly, a process pi delivers a message m when it invokes deli(m). The delivery of a message m by a process pi can be seen as the fact of passing the message m to the upper layer where this process pi is (the user pi in the case of the top layer). We omit the index i in these primitives when the process pi that invokes these primitives is not important.

With bcasti(m) process pi asynchronously sends a message m to each process pk ∈ Π, and deli(m) reports to the invoking process pi that m is the received message which is delivered. To preserve the anonymity of the system, we also consider that delivering processes can not identify the channel through which a broadcast message is received.

In the literature is always considered that broadcast and delivered messages are unique. It is traditionally assumed that every broadcast message m includes the different sender’s process identity as part of the content of m to distinguish it from other messages ([25], [59], [80], [103]). Since in AAS[0] processes are anonymous, we have to consider that messages are not unique. Hence, in AAS[0] several instances of a same message m can be broadcast or delivered. Thus, it is more accurate to say that in AAS[0] process pi sends an instance of message m to each process pk ∈ Π when it invokes bcasti(m), and process pi is reported of the delivering of an instance of a message m when it invokes deli(m). To simplify, we abuse
of the notation and we only distinguish between an instance of a message and the message itself when it is absolutely necessary.

Let $B_i$ be the multi-set of all instances of messages broadcast by process $p_i$, and let $D_i$ be the multi-set of all instances of messages delivered by process $p_i$. Let $B$ be the multi-set of all instances of messages broadcast in the system, i.e., $B = \bigcup_{p_i \in \Pi} B_i$. Similarly, $D = \bigcup_{p_i \in \Pi} D_i$ is the multi-set formed by all instances of messages delivered in the system. Hence, for example, if we have the following five primitives with the same message $m$: $bcast_i(m)$, $bcast_j(m)$, $del_i(m)$, $del_j(m)$ and $del_k(m)$, then the multi-set $B$ has two instances of $m$ and $D$ has three instances. (i.e., $B = \{m, m\}$, and $D = \{m, m, m\}$).

We assume that broadcast and deliver primitives of $AAS[\emptyset]$ do not give any fault-tolerant guarantees if a process crashes. Specifically, if a process crashes while it is executing $bcast(m)$, $m$ can be received by any subset of processes, and, hence, $del(m)$ can be invoked only by this subset of processes. Therefore, the system $AAS[\emptyset]$, with these two communication primitives, offers an *unreliable broadcast* service.

We formally define RB as follows:

- **Integrity:** $\forall p_i \in \Pi, D_i \subseteq B$.
- **Validity:** $\forall p_i \in Correct, \bigcup_{p_j \in Correct} B_j \subseteq D_i$.
- **Agreement:** $\forall p_i, p_j \in Correct, D_i = D_j$.

### 3.3.1 The Algorithm of Reliable Broadcast in $AAS_{R_{n,t}}[\emptyset]$  

The algorithm of Figure 3.1 implements the RB service in an anonymous asynchronous system $AAS_{R_{n,t}}[\emptyset]$ independently of the number of faulty processes.

**Description of the algorithm**

We say that process $p_i$ RB-broadcasts an instance of message $m$ if it invokes $RB_{bcast_i}(m)$ (line 3). Similarly, we say that process $p_i$ RB-delivers an instance of message $m$ is it invokes $RB_{del_i}(m)$ (line 15).

When process $p_i$ invokes $RB_{bcast_i}(m)$, it sends a message $(m, seq_i[m])$ to every process of the system $AAS[\emptyset]$, such that $m$ is the instance of the message to spread, and $seq_i[m]$ is the $p_i$’s number of sequence number of $m$ (line 5). The variable $seq_i[m]$ allows each process $p_j$ to distinguish among several instances of $m$ RB-broadcast by process $p_i$ (initially, $seq_i[m]$ is 0, line 2).

When process $p_i$ invokes $RB_{del_i}(m)$, that is, message $m$ with number of sequence $s$ (line 6), it uses $count_{msg_i}[m, s]$ to increase the number of messages $m$ with the same number of
sequence \( s \) delivered by process \( p_i \) (line 7). Then, it sends \((ACK, m, s, count_msg_i[m,s])\) to every process of the system AAS[\(0\)] (line 8).

When process \( p_i \) delivers \((ACK, m, s, c)\) for the first time, i.e., instance of \( m \) with sequence number \( s \) and a counter \( c \) (line 9), it relays this message \((ACK, m, s, c)\) crashes. To avoid relaying one message indefinitely, lines 9-11 are only executed if this message is delivered for the first time (line 9).

In order to guarantee each message \( m \) is RB-delivered for the same times as RB-broadcast times, \( p_i \) uses \( exec_i[m,s] \) variable and the function \( apply_msg(m,s,c) \). The variable \( exec_i[m,s] \) records the \( p_i \)'s execution times of \( RB_{del}(m) \) (initially \( exec_i[m,s] \) is 0 (line 1)). Before execute the function of \( apply_msg(m,s,c) \), \( p_i \) has to check the received \( c \) is greater than its variable \( exec_i[m,s] \) (line 13). If ‘yes’, then process \( p_i \) execute \( RB_{del}(m) \) for \( c - exec_i[m,s] \) (line 14) times.

```
1 Init:  
2 arrays seq_i, exec_i and count_msg_i have 0 in all positions  

3 When RB_bcast(m) is executed:  
4 seq_i[m] ← seq_i[m] + 1  
5 bcast_i(m, seq_i[m])  

6 When del_i(m, s) is executed:  
7 count_msg_i[m, s] ← count_msg_i[m, s] + 1  
8 bcast_i(ACK, m, s, count_msg_i[m, s])  

9 When del_i(ACK, m, s, c) is executed for first time:  
10 bcast_i(ACK, m, s, c)  
11 apply_msg(m, s, c)  

12 function apply_msg(m, s, c):  
13 if (exec_i[m, s] < c) then  
14 for (j = exec_i[m, s] + 1 to c) do  
15 RB_del_i(m)  
16 end for  
17 exec_i[m, s] ← c  
18 end if  
```

Fig. 3.1 The algorithm of RB in AAS_Rn,t[\(0\)] (code of \( p_i \))

Correctness Proof of the Algorithm

**Lemma 3.1** Integrity: \( \forall p_i \in \Pi, D_i \subseteq B. \)

**Proof:** Let us consider, by the way of contradiction, that the claim is not true. Then, there is a process \( p_i \) such that \( D_i \supset B \). Following the contradiction, we have that \( RB_bcast(m) \) is
executed $x$ times, and $RB_{\text{del}}(m)$ is executed $y > x$ times. Note that in one extreme case
$x$ processes can execute $RB_{\text{bcast}}(m)$ once, and, in the other, a same process can execute
$RB_{\text{bcast}}(m) x$ times.

A process $p_k$ increments its local number of instance $s$ of $m$ by one (line 4) previously
to execute $bcast(m, s)$ (line 5). Then, for each process $p_k$, the values of $s$ for $m$ that are
broadcast are 1, 2, 3, · · ·. So, in this case, these values of $s$ for $m$ that are broadcast by any
process will be in the range from 1, 2, 3, · · · up to $x$. On the other hand, each time that a
process $p_k$ delivers a number of instance $s$ of $m$ executing $del_k(m, s)$ (line 6), it counts this
number of instances incrementing $\text{count}_{\text{msg}_k}[m, s]$ by one (line 7). Hence, because links
are reliable and neither duplicate nor create spurious messages, if $r \leq x$ processes execute
$bcast(m, s)$, then every correct process executes $bcast(m, s)$ (line 5). So, its values of $s$ for $m$
when $bcast(m, s)$ (line 15) when $del_i(ACK, m, s, c)$ is also executed, but if it has not been applied yet, i.e., if $c > \text{exec}_i[m, s]$ (line 13). Then,
$RB_{\text{bcast}}(m)$ is executed $x$ times, and, $RB_{\text{del}}(m)$ is executed $c$ times, being $c \leq x$. So, we
reach a contradiction, and, hence, $\forall p_i \in \Pi$, $D_i \subseteq B$.

Lemma 3.2 Validity: $\forall p_i \in \text{Correct}, \bigcup_{p_j \in \text{Correct}} B_j \subseteq D_i$.

Proof: A correct process $p_j$ increments its local number of instance $s$ of $m$ by one (line 4)
previously to execute $bcast(m, s)$ (line 5). So, its values of $s$ for $m$ that are broadcast are 1, 2,
3, · · ·.

On the other hand, each time that a correct process $p_j$ delivers a number of instance $s$ of $m$
execting $del_j(m, s)$ (line 6), it counts this number of instances incrementing $\text{count}_{\text{msg}_j}[m, s]$ by one (line 7). Hence, because links are reliable and neither duplicate nor create spurious
messages, if $c$ correct processes executes $bcat(m, s)$, then every correct process $p_j$ broadcast
the sequence of messages $bcast_j(ACK, m, s, 1), bcast_j(ACK, m, s, 2), \cdots, bcast_j(ACK, m, s, c)$. Thus, because links are reliable and neither duplicate nor create spurious messages,
every correct process $p_j$ eventually receives the messages of these broadcast primitives, and
executes their corresponding $del_i(ACK, m, -, -)$.
3.3 Fault-tolerant Broadcast in Anonymous Distributed Systems with Reliable Communication Channels

We can observe that any correct process $p_i$ stores in $\text{exec}_i[m, s]$ the number of invocations of $\text{RB}_{\text{del}}_i(m)$ for each instance $s$ of $m$ when $\text{del}_i(\text{ACK}, m, s, c)$ is executed (lines 14-17). We can also observe that process $p_i$ only delivers the instance $s$ of $m$, executing $\text{RB}_{\text{del}}_i(m)$, when $\text{del}_i(\text{ACK}, m, s, c)$ is also executed, but if it has not been applied yet (line 13). Hence, if $\text{RB}_{\text{bcast}}(m)$ is executed $x$ times, such that $c$ of them are correct processes, then $\text{RB}_{\text{del}}_i(m)$ is executed at least $c$ times. Hence, $\text{RB}_{\text{del}}_i(m)$ is executed at least $c$ times. We complete the proof of Lemma 3.2. □

Lemma 3.3 Agreement: $\forall p_i, p_j \in \text{Correct}, D_i = D_j$.

Proof: Let us consider, by the way of contradiction, that the claim is not true. Following the contradiction, let us consider, w.l.o.g., that a correct process $p_i$ RB-delivers $x$ instances of a message $m$, and a correct process $p_j$ RB-delivers $x' < x$ instances of this message $m$.

If correct process $p_i$ RB-delivers $x$ instances, it also executes $x$ times $\text{del}_i(\text{ACK}, m, -, c)$ and, such that $c \leq x$ (lines 12-18), and hence, it also executes their corresponding $\text{bcast}_i(\text{ACK}, m, -, c)$ (lines 10, 11). Note that $c \leq x$ because each process increments $\text{count}_{msg}_j[m, s]$ by one (line 7), and links are reliable and not duplicate or create spurious messages. After all this happens, correct process $p_j$ will also eventually execute $x$ times the primitive $\text{del}_j(\text{ACK}, m, -, c)$, being $c \leq x$ (line 9), and process $p_j$ will eventually have to deliver from $x'$ to $x$ instances of $m$ (lines 14-17). Therefore, we reach a contradiction, and $\forall p_i, p_j \in \text{Correct}, D_i = D_j$. □

Theorem 3.1 The algorithm in Figure 3.1 guarantees the properties of RB.

Proof: According to Lemma 3.1, 3.2 and 3.3, it is easy to see that Theorem 3.1 is correct. □

3.3.2 The Algorithm of URB in $\text{AAS}_{R_{n,t}}[t < n/2]$

A URB implementation algorithm in $\text{AAS}_{R_{n,t}}[t < n/2]$ is described as Algorithm in Figure 3.2. As far as we know, implementing the URB abstraction in the classic asynchronous systems with a majority of correct processes $\text{AS}_{R_{n,t}}[t < n/2]$ is trivial. The construction relies on one condition: a message $m$ has been received by at least one non-faulty process, then this $m$ can be locally URB-delivered to the upper application layer. As $n > 2t$, this means that, without risking to be blocked forever, a process may URB-deliver $m$ as soon as it knows that at least $t + 1$ processes have received a copy of $m$. In fact, this condition is also needed to be satisfied in anonymous system.

In general, the identical message may be caused either by (1) the channel duplicating or (2) process sends two messages with the same value. The first kind of identical message
must be discarded, furthermore, it also cannot appear in our system model due to the reliable channel. In contrast, the second kind of identical message is meaningful, we have to deal with it. However, this kind of identical message also appears in classic systems, the way for the communication protocols to distinguish them is to attach a sequence number \( Seq[m] \) to each message. This idea also can be used in anonymous systems. But only one sequence number 0 is needed. Why? In anonymous systems, all identical messages are treated as a group, 0 is the symbol of a new independent message (original message) that should be counted. It is only needed to count the total number of these identical messages (0 symbols).

Remind the validity and uniform integrity property of URB requires that each anonymous identical message has to be delivered and one message deliver once. This will be satisfied well as if we count the total number of identical message and deliver each message for the total number times as we distinguish them one by one precisely and deliver each message once. So, we use the counting technique to count the identical messages.

The counting technique uses a pair of variables \((m, count)\), and two kinds of messages have to be defined previously as follows:

* **Original message**: a message comes from its sender directly that it has not been counted (received) before. We use \( count = seq[m] = 0 \) to denote it.

* **Forwarded message**: message comes from an intermediary. Denoted by a non-zero \( count \), \( count \neq 0 \) also means \( m \) has been received \( count \) times.

Once a process receives an original message, it has to count one time, then forwards this message with its count. When a process receives a forwarded message, if it is the first time receives this message, then just forward it; if is not the first time, then keeps silent.

**Description of the Algorithm**

The algorithm in Figure 3.2 is basically the same as algorithm in Figure 3.1 except when process \( p_i \) executes \( del_i(ACK, m, s, c) \) (lines 9-16). In algorithm of Figure 3.2, process \( p_i \) also relays this message \( (ACK, m, s, c) \) when it is executed by \( p_i \) for first time (line 10-12). To preserve the necessity of a majority of correct processes, process \( p_i \) uses \( count_{ack}[m, s, c] \) to count the number of messages \( (ACK, m, s, c) \) received it (line 13). If process \( p_i \) has received a message \( (ACK, m, s, c) \) from a majority of processes, then \( p_i \) applies this message (lines 14-16), executing \( URB_{del_i}(m) \), from next time, indicated in \( exec_i[m, s]+1 \), until the value of the counter of messages \( (m, s) \), indicated by \( c \) (lines 19-22). Similarly to algorithm of Figure 3.1, to avoid to URB-deliver messages due to outdated delivery of messages \( (ACK, m, s, c) \), the value \( c \) has to be greater than \( exec_i[m, s] \) (line 18).

**Lemma 3.4** Validity: \( \forall p_i \in Correct, \bigcup_{p_j \in Correct} B_j \subseteq D_i \).
3.3 Fault-tolerant Broadcast in Anonymous Distributed Systems with Reliable Communication Channels

![Algorithm](image)

**Proof:** A correct process $p_j$ increments its local sequence number of instance $s$ of $m$ by one (line 4) previously to execute $bcast(m, s)$ (line 5). So, its value of $s$ for $m$ that are broadcast are $1, 2, 3, \ldots$

On the other hand, each time that a correct process $p_j$ delivers a number of instance $s$ of $m$ executing $del_i(m, s)$ (line 6), it counts this number of instances incrementing $count_{msg}[m, s]$ by one (line 7). Because the communication channels are reliable and neither duplicate nor create spurious messages, if $c$ correct processes execute $bcast(m, s)$, then every correct process $p_j$ broadcasts the sequence of messages $bcast_j(ACK, m, s, 1), bcast_j(ACK, m, s, 2), \ldots, bcast_j(ACK, m, s, c)$. For reliable channels and a majority of correct processes are correct, every correct process $p_i$ eventually receives the messages of these broadcast primitives from a majority of processes, and it executes its corresponding line 15. As $p_i$ stores in $exec_i[m, s]$ the number of invocations of $URB\_del_i(m)$ for each instance $s$ of $m$ when $apply\_msg(m, s, c)$ is executed, and process $p_i$ only $URB\_delivers$ the instance $s$ of $m$ if it has not been applied yet (line 18), hence, $p_i$ $URB\_delivers$ $m$ at least $c$ times because $URB\_bcast(m)$ is executed at least $c$ times.
According all above, we complete the proof of Lemma 3.4.

**Lemma 3.5 Agreement:** For all \( p_i, p_j \in \text{Correct} \), \( D_i = D_j \).

**Proof:** Let us consider, w.l.o.g., that a correct process \( p_i \) URB-delivers \( x \) instances of a message \( m \), and a correct process \( p_j \) URB-delivers \( x' < x \) instances of this message \( m \).

If correct process \( p_i \) URB-delivers \( x \) instances of \( m \), it eventually executes line 20 of function \( \text{apply}_\text{msg}() \) with parameters \( (m, -, c_1), \ldots, (m, -, c_s) \) such that \( c_1 + \ldots + c_s = x \). To do so, process \( p_i \) has to receive each corresponding message \( (\text{ACK}, m, -, c_1), \ldots, (\text{ACK}, m, -, c_s) \) from at least a majority of processes (line 14-16). Then, a majority of processes executes \( \text{bcast}_i(\text{ACK}, m, -, c_1), \ldots, \text{bcast}_i(\text{ACK}, m, -, c_1) \), and each one of them re-broadcasts these messages the first time they receive them (lines 10-12). Because channels are reliable and a majority of processes are correct, all these \( x \) messages \( (\text{ACK}, m, -, c_1), \ldots, (\text{ACK}, m, -, c_s) \) will be received by correct process \( p_j \), and it eventually also has to URB-deliver from \( x' \) to \( x \) instances of \( m \) (line 20).

Hence, we can see that Lemma 3.5 is correct.

**Lemma 3.6 Integrity:** \( \forall p_i \in \Pi, D_i \subseteq B \).

**Proof:** The proof is similar to Lemma 3.1.

**Lemma 3.7 Uniformity:** \( \forall p_i \in \text{Faulty}, \text{and } p_j \in \text{Correct}, D_i = D_j \).

**Proof:** Each time that a faulty process \( p_i \) URB-delivers \( m \), it executes line 20 into the function \( \text{apply}_\text{msg}() \) with parameters, w.l.o.g. \( (m, s', c') \). Note that this happens because process \( p_i \) has received the message \( (\text{ACK}, m, s', c') \) from a majority of processes. Hence, process \( p_i \) also has to receive \( (\text{ACK}, m, s', c') \) from a majority of processes (line 14-16). Then, a majority of processes executes \( \text{bcast}_i(\text{ACK}, m, s', c'), \) and each one of them re-broadcasts this message the first time they receive it (lines 10-12). Thus, because channels are reliable and a majority of correct processes, this message \( (\text{ACK}, m, s', c') \) will be received by correct process \( p_j \), and it eventually also has to deliver at least \( c' \) instances of \( m \) (lines 19-21). Therefore, \( \forall p_i \in \text{Faulty}, \text{and } p_j \in \text{Correct}, D_i = D_j \).

**Theorem 3.2** The algorithm guarantees the properties of URB.

**Proof:** According to Lemma 3.4, Lemma 3.5, Lemma 3.6, and Lemma 3.7, it is easy to see that Theorem 3.2 is correct.
3.3 Fault-tolerant Broadcast in Anonymous Distributed Systems with Reliable Communication Channels

3.3.3 Impossibility to Implement URB in $AAS_{R_{n,t}}[t > n/2]$

In this section we prove that a majority of correct process is a necessary condition to solve URB in anonymous asynchronous system without additional assumption.

**Theorem 3.3** There is no algorithm $A$ that implements URB in every run of anonymous asynchronous systems when a majority of processes can crash (i.e., $AAS_{R_{n,t}}[t > n/2]$), and when processes do not know the maximum number of faulty processes (i.e., $t$ is unknown).

**Proof:** The proof is by contradiction, let us assume that there is an algorithm $A$ that implements URB in every run of $AAS_{R_{n,t}}[t > n/2]$, and when processes do not know $t$. Let us consider the following two valid runs $R_1$ and $R_2$ of $A$.

In $R_1$, a correct process $p_b$ executes the URB broadcast primitive with the message $m$ and to preserve the Validity Property of URB, a correct process $p_d$ executes at time $\tau$ the URB deliver primitive with the message $m$. We consider that $p_d$ delivers $m$ after receiving $x$ messages acknowledging that $x$ processes have also delivered this message $m$. Note that a process only knows that in a run the rest of processes can crash, but it does not know how many of them will crash in $R_1$ or who will be. So, $x$ is not related to the number of correct processes. Finally, we consider that in $R_1$ the transmission of any other message not previously specified is delayed in this asynchronous system until time $\tau'$ ($\tau' > \tau$).

$R_2$ is the same execution of $R_1$ until time $\tau$. So, $R_1$ and $R_2$ are indistinguishable until time $\tau$. Then, let us consider in $R_2$ that $p_b$ crashes at time $\tau''$ ($\tau < \tau'' < \tau'$). We also consider in $R_2$ that $p_d$ crashes at time $\tau''$, hence, after delivering $m$. Similarly, let us consider that these $x$ processes, that informed $p_d$ about their delivery of $m$, also crash at time $\tau''$. Note that, as process $p_d$ does not know a priori anything about correct processes, it can happen that the intersection between the set of these $x$ processes and the set pf correct processes can be empty. We also assume in $R_2$ that all transmitted messages in $R_1$ sent by faulty processes that were delayed until time $\tau'$ are lost in $R_2$. Note that, this can happen because reliable channels only guarantee the delivery of messages if sender and receiver processes are correct. Then, after time $\tau'$, there is no correct process in $R_2$ that has received any message related to $m$. Hence, we reach a contradiction, and there is a process $p_d$ that delivers $m$ at time $\tau$, but there is no correct process that can deliver $m$ in $R_2$ (which violate the Uniformity Property of URB).

Therefore, there is no algorithm $A$ that implements URB in every run of $AAS_{R_{n,t}}[t > n/2]$ when processes do not know $t$ in priori.

We complete the proof of Theorem 3.3. □
3.3.4 The Algorithm of URB in $AAS_{R_{n,t}[\psi]}$

Note that, due to the impossibility result, we need to use a failure detector [50] to enhance the system and circumvent this impossibility. Thus, the implementation of the URB in $AAS_{R_{n,t}}$ is possible for whatever the number of correct processes. In this section, we use failure detector $\psi$ proposed in [100]. An implementation algorithm of URB with $\psi$ is also given.

Failure detector $\psi$

Roughly speaking, the failure detector class $\psi$ returns at time $\tau$ a number $c$, such that $c$ is an upper bound of number of correct processes at time $\tau$ (transient period), but eventually $c$ converges towards exactly the number of correct processes (permanent period). Let us define $\psi$ more formally.

Definition 3.1 Let us consider that each process $p_i$ has a local variable $output_i$ that always returns an integer and positive value. We denote by $output_i^{\tau}$ this variable at time $\tau$. Let $|Correct|^\tau$ be the number of processes that are correct up to time $\tau$. For any process $p_i \in \Pi$ and run $R$, the variable $output_i$ must satisfy the following two properties:

- $\forall \tau, output_i^{\tau} \geq |Correct|^\tau$ (transient period).
- $\exists \tau: \forall \tau' \geq \tau, output_i^{\tau'} = |Correct|^{\tau'}$ (permanent period).

Description of the algorithm:

As in the algorithm of Figure 3.2, we say that a process $p_i$ URB-broadcasts an instance of message $m$ if it invokes $URB_{bcast_i}(m)$ (line 5), and URB-delivers an instance of message $m$ if $p_i$ invokes $URB_{del_i}(m)$ (line 25).

This algorithm in Figure 3.3 is similar to the algorithm in Figure 3.2. That main difference is that a process $p_i$, based on the value returned by the failure detector $\psi$ in $output_i$, has to wait until it delivers a number of messages ($ACK, m', s', c'$), indicated by $count_{ack}[m', s', c']$, broadcast by all correct processes. As the number of correct processes may change over time, process $p_i$ needs a task (task T2) where it can know this variation. In this task T2, process $p_i$ checks the variable $output_i$ of the failure detector permanently. Hence, if process $p_i$ delivered a number of messages ($ACK, m', s', c'$) at least equal to the current number of correct processes (line 18), it applies this message $m'$ in the same way that the algorithm of Figure 3.2 does (line 19, and lines 22-28).

Correctness Proof of the Algorithm:

Lemma 3.8 Validity: $\forall p_i \in Correct, \bigcup_{p_j \in Correct} B_j \subseteq D_i$. 
3.3 Fault-tolerant Broadcast in Anonymous Distributed Systems with Reliable Communication Channels

1 Init: 
2 arrays seq, exec, count_msg, and count_ack have 0 in all positions.
3 start tasks T1 and T2.

4 Task T1: 
5 When URB_bcast(m) is executed: 
6 seq[m] ← seq[m] + 1;
7 bcast(m, seq[m]).

8 When del_i(m, s) is executed: 
9 count_msg[m, s] ← count_msg[m, s] + 1;
10 bcast(ACK, m, s, count_msg[m, s]).

11 When del_i(ACK, m, s, c) is executed: 
12 if del_i(ACK, m, s, c) is executed for the first time then 
13 bcast_i(ACK, m, s, c);
14 end if
15 count_ack[m, s, c] ← count_ack[m, s, c] + 1.

16 Task T2: 
17 repeat forever
18 for each (count_ack_i[m', s', c'] ≥ output) do 
19 apply_msg(m, s, c);
20 end for
21 end if.

22 Function apply_msg(m, s, c): 
23 if exec_i[m, s] < c then 
24 for (j = exec_i[m, s] + 1 to c) do 
25 URB_del_j(m);
26 end for.
27 exec_i[m, s] ← c;
28 end if.

Fig. 3.3 The algorithm of URB in AAS_{Rn,\psi} (code of p_i)

Proof: A correct process p_j increases its local number of instance s of m by one (line 4) previously to execute bcast(m, s) (line 5). So, its value of s for m that are broadcast are 1, 2, 3, \ldots.

On the other hand, each time that a correct process p_j delivers a number of instance s of m executing del_j(m, s) (line 8), it counts this number of instances increasing count_msg_i[m, s] by one (line 9). Because the channels are reliable and neither duplicate nor create spurious messages, if c correct processes execute bcast(m, s), then every correct process p_j broadcasts the sequence of messages bcast_j(ACK, m, s, 1), bcast_j(ACK, m, s, 2), \ldots, bcast_j(ACK, m, s, c).

For reliable channels and a majority of correct processes are correct, every correct process p_i eventually has in its variable output_i of the failure detector \psi the number of correct processes.
(guaranteed by the failure detector), it eventually receives at least \( output_i \) messages (line 18), and hence, executing its corresponding line 19. As correct \( p_i \) stores in \( exec_i[m,s] \) the number of invocations of \( URB\_del_i(m) \) for each instance \( s \) of \( m \) when \( apply\_msg(m,s,c) \) is executed, and process \( p_i \) only \( URB\_delivers \) the instance \( s \) of \( m \) if it has not been applied yet (line 23), hence, \( p_i \) URB-delivers \( m \) at least \( c \) times because \( URB\_bcast(m) \) is executed at least \( c \) times.

According all above, we complete the proof of Lemma 3.8. \( \square \)

Lemma 3.9  Agreement: for all \( p_i, p_j \in Correct, D_i = D_j. \)

**Proof:** Let us consider, w.l.o.g., that a correct process \( p_i \) URB-delivers \( x \) instances of a message \( m \), and a correct process \( p_j \) URB-delivers \( x' < x \) instances of this message \( m \).

If correct process \( p_i \) URB-delivers \( x \) instances of \( m \), it eventually executes line 25 of function \( apply\_msg() \) with parameters \( (m, -, c_1), \ldots, (m, -, c_s) \) such that \( c_1 + \cdots + c_s = x \). To do so, process \( p_i \) has to receive each corresponding message \( (ACK, m, -, c_1), \ldots, (ACK, m, -, c_s) \) from at least \( output_i \) processes (line 18-20). Then, eventually \( output_i \) is equal to the number of correct processes (guaranteed by the failure detector), and it executes \( bcast_i(ACK, m, -, c_1), \ldots, bcast_i(ACK, m, -, c_s) \), and each one of them re-broadcasts these messages the first time they receive them (lines 12-14). Because channels are reliable and the variable \( output \) of the failure detector \( \psi \) of all correct processes eventually converges towards the number of correct processes (the definition of failure detector), all these \( x \) messages \( (ACK, m, -, c_1), \ldots, (ACK, m, -, c_s) \) will be received by correct process \( p_j \), and it eventually also has to URB-deliver from \( x' \) to \( x \) instances of \( m \) (line 24-26).

Hence, we can see that Lemma 3.9 is correct. \( \square \)

Lemma 3.10  Integrity: \( \forall p_i \in \Pi, D_i \subseteq B. \)

**Proof:** It is similar to the proof of Lemma 3.6. \( \square \)

Lemma 3.11  Uniformity: \( \forall p_i \in Faulty, and p_j \in Correct, D_i = D_j. \)

**Proof:** Each time that process \( p_i \) URB-delivers \( m \), it executes line 25 into the function \( apply\_msg() \) with parameters, w.l.o.g, \( (m, s', c') \). Thus, process \( p_i \) receives the message \( (ACK, m, s', c') \) from \( output_i \) processes (line 18-20). Then, eventually \( output_i \) is equal to the number of correct processes, and it executes \( bcast_i(ACK, m, s', c') \), and each one of them re-broadcasts this message the first time they receive it (lines 12-14). Thus, because channels are reliable and the variable \( output \) of all correct processes contains this exact number of correct processes, this message \( (ACK, m, s', c') \) will be received by correct process \( p_j \), and it eventually also has to deliver at least \( c' \) instances of \( m \) (lines 25). Therefore, \( \forall p_i \in Faulty, and p_j \in Correct, D_i = D_j. \) \( \square \)
3.4 Fault-tolerant Broadcast in Anonymous Distributed Systems with Fair Lossy Channels

**Theorem 3.4** The algorithm in Figure 3.3 guarantees the properties of URB.

**Proof:** According to Lemma 3.8, 3.9, 3.10, and 3.11, it is easy to see that Theorem 3.4 is correct.

**Remark 3.1** In this section, we have studied the fault-tolerant broadcast in anonymous systems. First, we include an implementation of the reliable broadcast (RB) for anonymous systems. On the possibility to implement the uniform reliable broadcast (URB) service, in this paper we prove the impossibility to implement the uniform reliable broadcast (URB) service when a majority of processes can crash and the amount of crashed processes is unknown by the correct processes, and the possibility of implement it when only a minority can crash. To extend the implementability of the URB service circumventing this impossibility result, we present an algorithm that implements the URB service in anonymous asynchronous systems independently of the number of crashed processes. We do it enriching the system with a failure detector (we use $\psi$ because it is a failure detector that works without knowing the identities of the processes).

3.4 Fault-tolerant Broadcast in Anonymous Distributed Systems with Fair Lossy Channels

Fault-tolerant Broadcast services are studied in anonymous distributed systems with fair lossy communication channels in this section. The anonymous asynchronous system is denoted as $AAS_F[n, t]$.

3.4.1 The Algorithm of Reliable Broadcast in $AAS_F[n, t][0]$ 

In this section, an algorithm of Reliable Broadcast in anonymous asynchronous systems with fair lossy channels is proposed firstly. This algorithm runs independently of the number of faulty processes. It is necessary to mention that it also satisfies the property of reliable broadcast if all processes are crashed.

The basic idea of our algorithm is intuitive that is try to make every message to be unique. As we all known, this idea is very easy to implement in non-anonymous systems, normally by using the identifier of a message’s sender and a sequence number. However, in anonymous systems, we cannot use this method due to that all processes do not have identifiers, and it is impossible to distinguish identical messages only by a sequence number. In order to deploy this idea in anonymous systems, we propose to use a random function. This random function can generate different random numbers and assigns one of them to a message as its unique
label, denoted by \textit{tag}. We assume that none random functions can generate a random number twice to two different messages. Adding a label \textit{(tag)} to each message instead of assigning an label to each process, the algorithm does not compromise of the privacy of the system. If a label is assigned to a process, this process can be tracked by detecting its flow of messages. Because this process uses the fixed label after it was assigned this label. If give a label to one message, there is no possibility to speculate the information of any process, because it is difficult to relate a message to its sender. All processes are still keeping anonymity to each other in the system.

Firstly, we give a precise description of the algorithm in Figure 3.4.

\textbf{Description of the algorithm:} Let’s consider a process \textit{p} \textit{i} to simplify the description of this algorithm (index \textit{i} is just used for description purpose, no process knows which process is \textit{p} \textit{i}, even itself). \textit{P} \textit{i} manages a random function \textit{random} \textit{i}() and two local sets:

- \textit{MSG} \textit{i}, initialized to empty, records all messages received by \textit{p} \textit{i}.
- \textit{RB\_DELIVERED} \textit{i}, initialized to empty, records all messages RBdelivered by \textit{p} \textit{i}.

This algorithm runs as follows: two sets of \textit{MSG} \textit{i} and \textit{RB\_DELIVERED} \textit{i} are initialized to empty firstly and activates Task 1 (lines 1-3). When \textit{p} \textit{i} calls \textit{RB\_broadcast} \textit{i}(\textit{m}) (line 4), \textit{random} \textit{i}() generates a random \textit{label (tag)} for \textit{m} firstly (line 5). Then, \textit{p} \textit{i} inserts \textit{(m, tag)} into \textit{MSG} \textit{i} (line 6), and this \textit{(m, tag)} will be broadcast periodically in Task 1 (lines 15-19).

When \textit{receive} \textit{i}(\textit{MSG, m, tag}) is executed (line 7), \textit{p} \textit{i} inserts \textit{(m, tag)} into \textit{MSG} \textit{i} if this is the first reception of \textit{m} (lines 8-10). Then, \textit{p} \textit{i} checks whether this \textit{m} has already been \textit{RB\_delivered} (line 11). If not, \textit{p} \textit{i} inserts \textit{(m, tag)} into \textit{RB\_DELIVERED} \textit{i} firstly and \textit{RB\_deliver} \textit{m} for one time (lines 12, 13).

Every message existed in \textit{MSG} \textit{i} is broadcast forever by \textit{p} \textit{i} in Task 1, overcoming the message lost caused by the fair lossy communication channels.

\textbf{Lemma 3.12} If a correct process \textit{RB\_broadcast} a message \textit{m}, then it eventually \textit{RB\_deliver} \textit{m}. (Validity)

\textbf{Proof:} Let us consider a non-fault process \textit{p} \textit{i} (\textit{i} is used for description, no process knows it in the system) that activates \textit{RB\_broadcast} \textit{(m)}. It calculates a unique random number as a \textit{tag} for this message \textit{m} firstly (Line 5), then insert \textit{(m, tag)} into the set \textit{MSG} \textit{i} to broadcast to all process (included itself) (Lines 6, 15-19). For \textit{p} \textit{i} is correct, this Task 1 will execute forever to diffuse messages. Together with the fairness property of fair lossy channel, \textit{p} \textit{i} will receive \textit{m} eventually. Then, \textit{p} \textit{i} will \textit{RB\_deliver}() \textit{m}, because it’s the first time of receiving \textit{m} and this \textit{m} has not been \textit{RB\_delivered} before (Lines 11-14). We finish the proof of this Lemma 3.12.
3.4 Fault-tolerant Broadcast in Anonymous Distributed Systems with Fair Lossy Channels

1 Initialization
2 sets \( MSG_i, RB\_DELIVERED_i \) empty
3 activate Task 1

4 When \( RB\_broadcast_i(m) \) is executed
5 \( \text{tag} \leftarrow \text{random}() \)
6 insert \((m, \text{tag})\) into \( MSG_i \)

7 When \( receive_i(MSG, m, \text{tag}) \) is executed
8 if \((m, \text{tag})\) is not in \( MSG_i \) then
9 insert \((m, \text{tag})\) into \( MSG_i \)
10 end if
11 if \((m, \text{tag})\) is not in \( RB\_DELIVERED_i \) then
12 insert \((m, \text{tag})\) into \( RB\_DELIVERED_i \)
13 \( RB\_deliver_i(m) \)
14 end if

Task 1:
15 repeat forever
16 for every message \((m, \text{tag})\) in \( MSG_i \) do
17 broadcast\((MSG, m, \text{tag})\)
18 end for
19 end repeat

Fig. 3.4 The algorithm of RB in \( AAS_{F_n,\emptyset} \) (code of \( p_i \))

Lemma 3.13 If a correct process \( RB\_deliver \) a message \( m \), then all correct process eventually \( RB\_deliver \) \( m \). (Agreement)

Proof: Let us consider, by the way of contradiction, that the claim is not true. It means that if a correct process \( p_i \) has delivered a message \( m \), exists at least one correct process does not deliver it.

In fact, according to the algorithm (Line 6), \( m \) has to be saved in the set \( MSG_i \) of \( p_i \) when \( RB\_broadcast() \) is called. Then, \( p_i \) executes Task 1 that to send every message that existed in its set \( MSG_i \) forever, including \((m, \text{tag})\) (Lines 15-19). If there exists one correct process does not deliver this \( m \), this means that this correct process does not receive \( m \). We get a contradiction here. Because according the fairness property of the channel, all correct process will eventually receive \((m, \text{tag})\) and \( RB\_deliver() \) this \( m \) (Lines 8-14) for one time because \( p_i \) is a correct process and sends \( m \) forever. Hence, Lemma 3.13 is correct. \( \square \)

Lemma 3.14 For any message \( m \), every correct process \( RB\_deliver \) \( m \) at most once, and only if \( m \) was previously \( RB\_broadcast \) by sender\((m) \). (Integrity)

Proof: The second part of this lemma that any message \( m \) was previously \( RB\_broadcast \) by its sender is trivial, because each process only forward received messages and fair lossy channels do not create, duplicate, or garble messages.
Then, we focus on the proof of the first part of this lemma. We suppose in algorithm 1 that each message has a unique tag, and each process has a set $RB_{\text{DELIVERED}}_i$ to record all messages that have delivered (Line 12) to avoid $RB_{\text{deliver}}()$ a message more than one time. Even though each message can be sent forever and will be received by every correct process for infinite times (Lines 15-19), every message has to be checked whether it has already been delivered by checking if this message has already existed in its set $RB_{\text{DELIVERED}}_i$ (Line 11). It is certain that no message $m$ will be delivered more than once. We finish the proof of Lemma 3.14.

**Theorem 3.5** Algorithm of Figure 3.4 guarantees the property of reliable broadcast.

**Proof:** According to Lemma 3.12, Lemma 3.13 and Lemma 3.14, it is trivial to see that Theorem 3.4 is correct.

**Remark 3.2** Different from the implementation of reliable broadcast in an anonymous distributed system with reliable communication channels, in this system model with fair lossy channel, every message that has been $RB_{\text{delivered}}$ by a correct process needs to be broadcast forever (Lines 15-19) in order to overcome the message lost caused by the channel. This forever broadcast can be considered as the cost of obtaining reliability over the unreliable fair lossy channels in anonymous asynchronous distributed systems. This forever broadcast is the reason that makes algorithm of Figure 3.4 to be a non-quiescent too.

Furthermore, the fair lossy channels is also the reason to use random tags. Reliable channels do not duplicate messages, so we can count equal messages knowing that they come from different application sends. However, fair lossy channels may duplicate internally in the channels a same message and the only way to identify them as belonging to the same message is because they have the same tag and content.

### 3.4.2 The Algorithm of Uniform Reliable Broadcast in $AAS_{Fn,t}[t<n/2]$}

In this section, an implementation algorithm of Uniform Reliable Broadcast is proposed. The system model used in this section has a difference from the previous part of Reliable Broadcast, i.e., an extra requirement of a majority of correct processes is needed in anonymous asynchronous distributed systems where processes are connected by fair lossy channels. The system model of this section is denoted by $AAS_{Fn,t}[t < n/2]$.

The definition of agreement property of URB is stricter than RB’s, it requires all correct processes $URB_{\text{deliver}}$ the same set of messages and the messages URB-delivered by crashed processes is a subset of messages that $URB – \text{delivered}$ by correct processes. As
in non-anonymous distributed systems, a process can \textit{URB\_deliver} a message when it is
guaranteed that at least one correct process has received this message. This condition is also
required in the anonymous asynchronous distributed systems. However, there is no easy
way to confirm the correct process who has received one message in anonymous distributed
systems, because all processes have no identifiers. To conquer this difficulty, the intuition of
implementation of URB in \textit{AAS\_F\_n[t < n/2]} is as follows: 1) to add a label (\textit{tag}) to each
message like the algorithm in Figure 3.4; 2) to add an acknowledge message (denoted by
\textit{ACK}), and a corresponding unique \textit{tag2} to each acknowledge message.

The \textit{URB\_deliver} condition can be expressed as follows: each process can \textit{URB\_deliver}
a message \(m\) if it has received a majority of distinct \textit{ACKs} of this \(m\). This condition together
with a majority of correct processes in the system guarantee that at least one correct process
has received this message \(m\), which implies that all correct processes will eventually receive
and \textit{URB\_deliver} \(m\).

\textbf{Description of the algorithm}: Algorithm in Figure 3.5 is the implementation algorithm of
Uniform Reliable Broadcast in \textit{AAS\_F\_n[t < n/2]}. In this algorithm, two kinds of messages
will be transmitted: \textit{MSG} (a message needs to be \textit{URB\_delivered}) and \textit{ACK} (reception
acknowledge of one message). Each process manages a random function and four local sets:

- \(MSG_i\), initialized to empty, that records all messages that it has received.
- \(URB\_DELIVERED_i\), initialized to empty, that records all \textit{URB\_delivered} messages.
- \(MY\_ACK_i\), initialized to empty, that records all acknowledge messages of each message
generated by itself.
- \(ALL\_ACK_i\), initialized to empty, that records all acknowledge messages of each mes-
sage it has received (generated by any process).

Let us consider a process \(p_i\). At the beginning, \(p_i\) initializes all its four local sets into
empty and activates Task 1 (Lines 1-3).

When \(p_i\) calls \textit{URB\_broadcast}(\(m\)), like the algorithm of Reliable Broadcast, \(p_i\) generates
a unique random number as a \textit{tag} of this message \(m\), then inserts this pair of \((m,\text{tag})\) into
\(MSG_i\) (Lines 4-6). After that, \(p_i\) begins the Task 1 to propagate each message that already
existed in the set of \(MSG_i\) forever (Lines 28-32).

When \(p_i\) receives a message \((MSG,m,\text{tag})\) (may come from itself or other process), there
are three cases:

- If \(p_i\) receives \((MSG,m,\text{tag})\) from itself (i.e. if this \((m,\text{tag})\) has already existed in
  \(MSG_i\) and its ACK message \((m,\text{tag},\text{tag2})\) does not exist in \(MY\_ACK_i\)) for the first
Fig. 3.5 The algorithm of URB in AAS\(_{F_n,t}[0]\) (code of \(p_i\))

time. This process will go to execute line 14, generates a random number (\(tag2\)) to tag the acknowledge message of \((m,tag)\). Then, \(p_i\) saves the acknowledge message \((m,tag,tag2)\) into its sets \(MY\_ACK_i\), and sends this acknowledgment message \((ACK,m,tag,tag2)\) to all processes to confirm its reception of \((m,tag)\) (Lines 15,16). This \(tag2\) is unique for each pair of \((m,tag)\), which means \(tag2\) cannot be changed for the same pair of \((m,tag)\) once it is generated. The local set \(MY\_tag\) is used to maintain this uniqueness, to distinguish the \(tag2\) generated by itself from others \(tag2\) that received from others process.
• If $p_i$ receives $(MSG, m, tag)$ from other process (i.e. if this $(m, tag)$ is not its $MSG_i$, neither its ACK message $(m, tag, tag2)$ does not exist in $MY\_ACK_i$ ) for the first time, it saves this message into $MSG_i$ (Lines 8, 9). Then, like the first case, $p_i$ generates a random number $(tag2)$ to tag the acknowledge message of $(m, tag)$ (Line 14). Then, $p_i$ saves the acknowledge message $(m, tag, tag2)$ into its sets $MY\_ACK_i$, and sends this acknowledgment message $(ACK, m, tag, tag2)$ to all processes to confirm its reception of $(m, tag)$ (Lines 15,16).

• If $p_i$ has received a $(m, tag)$ already (i.e. if this $(m, tag)$ has already existed in $MSG_i$ and its ACK message $(m, tag, tag2)$ also exists in $MY\_ACK_i$), it just re-send the corresponding acknowledge message $(ACK, m, tag, tag2)$ to all processes in order to confirm its reception of $(m, tag)$ to tolerate failures in the fair lossy channel (Lines 11,12).

When $p_i$ receives an acknowledge message (labeled by ACK) for the first time, it inserts this ACK message to its set $ALL\_ACK_i$ (Lines 19-21).

When $p_i$ receives a majority of acknowledge messages $(m, tag, −)$ (more than $n/2$ different tag2), and this $m$ with tag has not been $URB\_delivered$ yet, then $p_i$ executes $URB\_deliver(m)$ (Lines 22-25).

**Lemma 3.15** If a correct process $URB\_broadcasts$ a message $m$, then it eventually $URB\_deliver m$. (Validity)

**Proof:** Like the proof of Lemma 3.12, let us consider a non-fault process $p_i$ (i is used for description purpose, no process knows it in the system) that calls $URB\_broadcast(m)$. A unique random number is given to this message $m$ as a tag (Line 5), then insert $(m, tag)$ into the set $MSG_i$ to broadcast forever by Task 1 (Lines 28-32). According to the fairness property of fair lossy channel and a majority of correct processes, all correct processes (include $p_i$) will receive this $m$ eventually.

When a correct process receives $(MSG, m, tag)$ for the first time, it generates a second unique tag2 as the label of the corresponding acknowledgment message. Then the receiver sends this acknowledgment message to all processes. Because every process that has received $(MSG, m, tag)$ will broadcast forever and together with the fairness property of the channel, each correct process will receive this $MSG$ an infinite times. Due to that when a process receives a $MSG$ message, it will broadcast an acknowledgment to all. Then, each acknowledgment message will also be broadcast an infinite times. Eventually, $p_i$ will receive at least a majority of acknowledgment messages of $m$, and $URB\_deliver m$. We complete the proof of Lemma 3.15.
Lemma 3.16  If some process URB_deliver a message m, then all correct processes eventually URB_deliver m. (Uniform Agreement)

Proof:  To prove this lemma, we consider the following two cases:

Case 1: A message m is URB_delivered by a correct process.

A correct process $p_i$ URB_deliver a message m means that $p_i$ has received a majority of acknowledgment messages of this m, which also means this $p_i$ has received this m (Lines 7-10). According to lines 28-32, $p_i$ will execute Task 1 forever and forward m to all processes. With the fairness property of the communication channel, all correct processes will eventually receive m.

When a correct process receives (MSG, m, tag) for the first time, it acknowledges the reception by broadcast a unique (ACK, m, tag, tag2) (Lines 14-16). Together with a majority of correct processes assumption, it is obvious that each correct process will receive a majority of acknowledgment messages of m. Then, all of them will URB_deliver this message m for one time (Lines 22-25).

Case 2: A message m is URB_delivered by a crashed process.

Message m has been URB_delivered by a crashed process, which means that this process has received a majority of acknowledgment message (ACK, m, tag, tag2) before URB_deliver(m). According to the algorithm, only process who has received (MSG, m, tag) can send an acknowledgment message. So, the reception of a majority of acknowledgment messages of m is equivalent to the fact that a majority of processes have received the message m. Together with the assumption that a majority of processes are correct, it is obvious that at least one correct process has received (MSG, m, tag) before this crashed process URB_deliver m. Then, this correct process will broadcast m. Consider the proof of Lemma 3.15 and the case 1 of this lemma, it is trivial to know that all correct processes will URB_deliver m.

Following both case 1 and 2, we can see that Lemma 3.16 is correct. □

Lemma 3.17  For every message m, every process URB_deliver m at most once, and only if m was previously URB_broadcast by sender(m). (Uniform Integrity)

Proof:  Like the proof of Lemma 3.14, it is trivial to see that any message m was previously URB_broadcast by its sender. Because each process only forwards messages it has received and the fair lossy channel does not create, duplicate, or garble messages.

Then, we focus on the proof that each message only can be URB_delivered at most one time. Let us observe that every message has two unique tags: one is used to label the message itself; one is used to label the acknowledgment of this message. With these two tags, each
message or acknowledgment message can be distinguished. Together with the fact that each process has a \( MY_{\text{ACK}} \) set, which is used to record the mapping relationship between the received message and the generated tag (Lines 11, 12); \( URB_{\text{DELIVERED}} \) set is used to record all \( URB_{\text{delivered}} \) messages (Line 24).

Even though each message has to be broadcast forever and will be received by every correct process for an infinite times (Lines 28-32), with all mechanisms above, it is obvious that no message \( m \) will be \( URB_{\text{delivered}} \) more than once. Hence, the proof is completed.

\[ \square \]

**Theorem 3.6** The algorithm in Figure 3.5 guarantees the property of URB.

**Proof:** According to Lemma 3.15, 3.16 and 3.17, it is easy to see that Theorem 3.6 is correct.

\[ \square \]

**Remark 3.3** This URB algorithm is non-quiescent because it requires all correct processes to broadcast all its \( URB_{\text{delivered}} \) messages forever. However, it is worth to mention that this algorithm can fulfill a fast \( URB_{\text{deliver}}(\) of a message. This can happen due to the property of fair lossy communication channels and the asynchrony of the system. A process may receive an acknowledgment of a message earlier than the message itself. Then, when a process receives a majority of acknowledgment \( (\text{ACK}, m, \text{tag}, \text{tag2}) \) of a message \( m \), it \( URB_{\text{deliver}} \) this message \( m \). In fact, this faster \( URB_{\text{deliver}} \) does not violate the uniform property of URB, even if this faster deliver process crashes after \( URB_{\text{deliver}} \) m. Because this faster deliver process must have received a majority of acknowledgment messages of \( m \) before \( URB_{\text{deliver}}(\) it, which means a majority of processes have received the message (one process generates only one unique ACK message to a message). Together with the fact that there is a majority of correct processes in the system, it is obvious that at least one correct process has received this message \( m \). Then, this correct process will broadcast \( m \) forever that guarantees all correct processes will receive \( m \).

Another point needs to be mentioned is that it is necessary to generate a unique random number to every message in each set of messages (MSG or ACK), but one random number can be used by two different types of messages (i.e., one comes from MSG, one comes from ACK). Because we use MSG and ACK as a symbol when broadcasting messages, two message can be distinguished even they use the same random number.

### 3.4.3 An Impossibility Result

In this section, it is proved that a majority of correct processes is a necessary condition to implement the *Uniform Reliable Broadcast* (URB) abstraction in anonymous asynchronous
message passing distributed systems without any others additional assumption.

**Theorem 3.7** It is impossible to solve URB in AAS[0] without a majority of correct processes.

**Proof:** The proof is by contradiction, let us suppose that there exists an algorithm A solves URB in AAS[t ≥ n/2]. Then, all processes in the system are divided into two subsets $S_1$ and $S_2$, such that $|S_1| = \lceil n/2 \rceil$ and $|S_2| = \lfloor n/2 \rfloor$. Now, we consider two runs: $R_1$ and $R_2$.

- Run $R_1$. In this run, all processes of $S_2$ crash initially, and all the processes in $S_1$ are non-faulty. Moreover, if a process in $S_1$ issues $URB\_broadcast(m)$. Due to the very existence of the algorithm A, every process of $S_1$ will $URB\_deliver m$.

- Run $R_2$. In this run, all processes of $S_2$ are non-faulty, and no process of $S_2$ ever issues $URB\_broadcast()$. The processes of $S_1$ behave as in $R_1$: a process issues $URB\_broadcast(m)$, and they all $URB\_deliver m$. Moreover, after it has $URB\_delivered m$, every process of $S_1$ crashes, and all messages ever sent by the process of $S_1$ are lost, neither has been received by a process of $S_2$. Hence, no process in $S_2$ will $URB\_deliver m$.

It is easy to see that all processes of $S_1$ cannot distinguish run $R_2$ from run $R_1$ before they $URB\_deliver m$, as they did in run $R_1$. Then after that all processes in $S_1$ are crashed, together with the fair lossy channel, no process in $S_2$ has received $m$. This violates the uniform agreement of URB, so the algorithm A does not exist. We complete the proof of Theorem 3.7.

**3.4.4 Anonymous Failure Detector Class of $A\Theta$**

Following the impossibility result, one question appears naturally, that is, what extra information is needed if Uniform Reliable Broadcast abstraction is implemented under the assumption that any number of processes can crash? The answer is that the confirmation of a message $m$ has been received by at least one correct process $p_j$ before a process $p_i (i \neq j)$ $URB\_deliver$ this $m$. Thanks to the failure detector proposed by T. D. Chandra et al. [50], the confirmation can be guaranteed by the failure information (even unreliable) provided by it. Then comes another question that which failure detector is the weakest one for Uniform Reliable Broadcast (i.e., the minimum information needed from the failure detector). This will be discussed in this section.
The definition of $A\Theta$

In non-anonymous systems, failure detector $\Theta$ is proved [6] as the weakest one that it always trust at least one correct process (accuracy) and eventually every correct process do not trust any crashed process (completeness). The counterpart of $\Theta$ in anonymous distributed system $A\Theta$ is considered as the weakest failure detector.

$A\Theta$ provides the same information as $\Theta$ if process has identifier. However, it is impossible to give such information in anonymous systems because each process has no identifier. So, the key point to define and implement $A\Theta$ is how to identify a process without breaking anonymity of the system. We are inspired by the definition of failure detector class of $A\Sigma$, which is proposed by F. Bonnet and M. Raynal [41], to define the anonymous failure detector class of $A\Theta$. This $A\Theta$ provides each process with a read-only local variable $a_{\theta i}$ that contains pairs of $(\text{label}, \text{number})$, in which one label represents a temporary identifier of one process and number represents the number of correct processes who have known this label. Label is assigned randomly to each process which does not break the anonymity of system. Because each process does not know the mapping relationship between a label and a process (even itself).

The definition of $A\Theta$ is given as follows:

- $A\Theta$-completeness: There is a time after which variables $a_{\theta i}$ only contain pairs of $(\text{label}, \text{number})$ associated to correct processes.

- $A\Theta$-accuracy: If there is a correct process, then at every time, all pairs of $(\text{label}, \text{number})$ output by failure detector $A\Theta$ hold that every subset $T$ of size number of processes that know label contains at least one correct process (i.e., for each label, there always exists one correct process in any set of number processes that know this label).

Let us define $A\Theta$ more formally:

$$S(\text{label})=\{i \mid \exists \tau \in \mathbb{N}: (\text{label},-) \in a_{\theta i}^\tau\}.$$ $S(\text{label})$ is the set of all processes that know label at time $\tau$.

- $A\Theta$-completeness: $\exists \tau \in \mathbb{N}, \forall i \in \text{Correct}, \forall \tau' \geq \tau, \forall (\text{label},\text{number}) \in a_{\theta i}^\tau$: $|S(\text{label}) \cap \text{Correct}| = \text{number}.$

- $A\Theta$-accuracy: $\text{Correct} \neq \emptyset \implies \forall \tau \in \mathbb{N}, \forall i \in \Pi, \forall (\text{label},\text{number}) \in a_{\theta i}^\tau$: $\forall T \subseteq S(\text{label}), |T| = \text{number}$. $T \cap \text{Correct} \neq \emptyset.$
The Algorithm of $A\Theta$ in ASS[0]

In this section, the implementation algorithm of $A\Theta$ and the corresponding correct proof are given. The implementation is based on a synchronous system model with reliable communication channels that allows any number of crashed processes.

Description of the algorithm:

Each process has three sets and one array:

- Set $rec_i$, initialize to be empty, records the received label and the corresponding number of this label.
- Set $Output_i$, initialize to be $(0, n)$, is the output set of the failure detector.

In order to guarantee the accuracy property that there always exists one correct process in the output. Hence, every process initially has a label 0, and the number of this label is $n$ (the total number of processes in the distributed system). Then, a random label is assigned to each process (Line 4). Each process broadcasts, every $\beta$ time, its label as a heartbeat and set its timer $i$ to $\Delta$ (Lines 5-7). It is assumed that $\beta > \Delta$.

When process $p_i$ receives a heartbeat ($label_j$) from process $p_j$, it will do as follows:

If this $label_j$ has not existed in set $rec_i$ yet, $p_i$ insert ($label_j, 0$) into $rec_i$. Then, $p_i$ get a random number (tag of the ACK of this heartbeat) from the random function and broadcast ($label_j, ACK_i$)(Lines 9-14).

When process $p_i$ receives an ACK of a heartbeat ($label_j, ACK_i$) from $p_j$:

- If this $label_j$ does not exist in the set $rec_i$, then, insert ($label_j, 1$) into $rec_i$ (Lines 16, 17). This 1 means the Heartbeat of $label_j$ has been acknowledged by one process. Then, decrease the number of label 0 (Line 18).

- If this $label_j$ has already existed in $rec_i$, $p_i$ add one time to the number of $label_j$ in $rec_i$ (Line 20), and decrease the number of label 0.

When the timer $i$ expires, output the set $rec_i$. Then set $rec_i$ to be empty (Lines 20-22).

Correctness proof:

**Lemma 3.18** ($A\Theta$-completeness) There is a time after which variables $a_{\theta}$ only contain pairs of (label, number) associated to correct processes.

**Proof:** This completeness property of $A\Theta$ follows from the observation that, if a process $p_i$ crashes, it does not send its heartbeat any more, after $\Delta$ time, a correct process $p_j$ will receive none heartbeat from $p_i$. Hence, $p_j$ will not broadcast the ACK of $p_i$’s heartbeat anymore.
Then after $\Delta$ time, due to the synchrony of the channel, $p_i$ will be considered as crashed and be exempted from the output of $p_j$. Others correct process will follow the same way as $p_j$’s that they will not output the label of $p_i$ and its count number of ACK anymore. We complete the proof.

```
1 Initialization
2 set array $rec_i$ empty
3 $Output_i \leftarrow (0,n), rec_i(0,number_0) \leftarrow (0,n), n$ is the total number of processes
4 $label_i \leftarrow random_i()$
5 Repeat every $\beta$ time units
6 $broadcast_i(label_i)$
7 set timer$_i$ to $\Delta$
8 end repeat
9 Repeat every $\beta$ time units
10 $broadcast_i(label_i)$
11 set timer$_i$ to $\Delta$
12 end repeat
13 When $receive_i(label_j)$ from $p_j$ is executed
14 if $(label_j, -)$ is not in $rec_i$ then
15 insert $(label_j, 0)$ into $rec_i$
16 end if
17 $ACK_i \leftarrow random_i()$
18 $broadcast_i(label_j, ACK_i)$
19 When $receive_i(label_j, ACK_j)$ from $p_j$ is executed
20 if $(label_j, -)$ is not in $rec_i$ then
21 insert $(label_j, 1)$ into $rec_i$
22 replace $(0, number_0)\leftarrow(0, number_0 - 1)$ in $rec_i$
23 else
24 replace $(label_j, v)$ by $(label_j, v + 1)$ in $rec_i$
25 replace $(0, number_0)\leftarrow(0, number_0 - 1)$ in $rec_i$
26 end if
27 When timer$_i$ expires:
28 $output_i \leftarrow rec_i$
29 empty set $rec_i$
```

Fig. 3.6 The algorithm of failure detector $A\Theta$ (code of $p_i$)

**Lemma 3.19** ($A\Theta$-accuracy) If there is a correct process, then at every time, all pairs of $(label, number)$ output by failure detector $A\Theta$ hold that every subset $T$ of size number of processes that know label contains at least one correct process (i.e., for each label, there always exists one correct process in any set of number processes that know this label).

**Proof:** There is a correct process $p_i$, then $p_i$ will broadcast its heartbeat $label_i$ every $\beta$ time units. Others processes (include itself) will broadcast ACK messages when they receive
the heartbeat (Lines 8-13). Together with reliable channel, the heartbeat of \( p_i \) and all ACK messages of its heartbeat can be received by all correct processes. Then, it will be counted correctly by each correct process and output. The channel cannot duplicate, modify or create any message, and each message has a unique tag. It is obvious that the label of \( p_i \) and the corresponding number will be included correctly in the output of each correct process. We complete the proof.

**Theorem 3.8** Algorithm of Figure 3.6 guarantees the property of failure detector \( A\Theta \).

**Proof:** According to Lemma 3.18 and Lemma 3.19, it is trivial to see that Theorem 3.8 is correct.

### 3.4.5 The Algorithm of Uniform Reliable Broadcast in \( AAS_{F_{n,t}}[A\Theta] \)

This section presents the Uniform Reliable Broadcast algorithm in anonymous asynchronous systems enriched with an anonymous \( A\Theta \) failure detector. With the failure information (though unreliable) from \( A\Theta \), this algorithm runs with the assumption that any number of processes can crash in the system.

**The idea and description of the algorithm**

The failure detector \( A\Theta \) gives the information \((label, number)\), which means that there are number processes that know the label. Furthermore, there exists at least one correct process among these number processes and eventually all of the number processes are correct processes.

The condition that a process \( p_i \) can uniform reliable broadcast a message \( m \) is: \( p_i \) receives number ACKs of this \( m \) from a quorum \( A \) and there is another quorum \( B \) of processes who have known this label, these two quorum must be equal (i.e., include the same processes).

According to the URB algorithm, a process \( p_i \) receives ACK message in the form of \((ACK, m, tag, tag2)\), where \( tag \) is the identifier of a message \( m \), and \( tag2 \) is the identifier of ACK message of this \( m \). Every process assigns a unique identifier \( tag2 \) to its ACK message of a message \( m \). Without failure detector, \( p_i \) has to receive a majority of ACKs with different \( tag2 \), because together with a majority of correct processes assumption guarantee that at least one correct process has received this message \( m \). If the failure detector \( A\Theta \) is added to take place of a majority of correct processes, due to its properties, the condition of \( URB\_deliver \) can be changed to when it receives number ACKs of this \( m \) are exactly from the same number processes that outputted by \( A\Theta \).
If check the two conditions, one inconsistent problem can be found. The failure detector $A\Theta$ only give the number information about which processes have known one label and at least one of them is correct. It is difficult to give concrete identifiers information of the correct processes in anonymous systems. In another side, the $URB - deliver$ condition requires that all number ACKS of message $m$ must come from the same quorum of processes that output by $A\Theta$. A shared and consistent identifier information is needed here to break this uncertainty, otherwise, inconsistent problem may occur. For example, suppose the output of process $p_1$’s failure detector $A\Theta_1$ is $(label,k)$, and this $k$ refers to $k$ processes: $p_1, p_{n-k+2}, p_{n-k+3}, ..., p_n$, where only $p_1$ is a fault process. Then $p_1$ can $URB\_deliver$ a message $m$ when it receives $k$ ACKs from $k$ different processes (e.g., $p_1, p_2, ..., p_k$). After $URB\_deliver(m)$, $p_1, p_2, ..., p_k$ are all crash. At this point, no correct process has received this $m$ and will not $URB\_deliver m$, which violates the uniform agreement property of URB. This case can happen at least at the beginning of the algorithm running, because at this time the failure detector’s output may include non-correct processes. We propose two ways to solve this uncertainty, one way is to assign a fixed label to each process at the initial period of running; the second way is each process has a shared memory between the URB application layer and the $A\Theta$ layer. In this section, we focus on the first way, and the second way will be discussed in the appendix A.

Each process has to read and add its failure detector’s information before broadcast an acknowledgment message. It waits all ACK messages from others process until receives number ACKs messages with the label, meanwhile, the output of its failure detector of label is $(label, number)$. This means at least one correct process that knows this label has received this message. Then can $URB\_deliver()$ this $m$.

**Description of the algorithm**

This algorithm in Figure 3.7 is similar to the simple $URB$ algorithm (algorithm in Figure 3.5) in the system with a majority of correct processes instead of a failure detector. The main change is how to broadcast an acknowledgment message when a process receives message $(m, tag)$. To be more clearly, this new change will be described in three cases as follows:

- If $p_i$ receives $(MSG, m, tag)$ from itself (i.e. if this $(m, tag)$ has already existed in $MSG_i$ and its ACK message $(m, tag, tag2)$ does not exist in $MY\_ACK_i$) for the first time. This case is the same as before. This process will go to execute line 15, generates a random number $tag2$ as a tag of the acknowledge message of $(m, tag)$. Then, insert $(m, tag)$ and its corresponding $tag2$ into the set $MY\_ACK_i$. $p_i$ reads labels from its failure detector $A\Theta_i$, and broadcasts $(ACK, m, tag, tag2, label_i)$ as an acknowledge message to all processes to confirm its reception of $(m, tag)$ (Lines 16-18). This $tag2$
Fault-tolerant Broadcast in Anonymous Distributed Systems

is unique for each pair of \((m, tag)\), which means \(tag2\) cannot be changed for a pair of \((m, tag)\) once it is generated. The local set \(MY\_ACK_i\) is used to distinguish the \(tag2\) generated by \(p_i\) itself from others \(tag2\) that received from others process.

- If \(p_i\) receives \((MSG, m, tag)\) from others process for the first time (i.e., if this \((m, tag)\) is not its \(MSG_i\), neither its ACK message \((m, tag, tag2)\) does not exist in \(MY\_ACK_i\)), it saves this message into \(MSG_i\) (Lines 8, 9). Then, like the first case, \(p_i\) generates a random number \(tag2\) as a tag of the acknowledge message of \((m, tag)\) (Line 15), and saves them in \(MY\_ACK_i\). Then, reads and adds the label information from its failure detector \(A\Theta_i\) into the acknowledge message \((ACK, m, tag, tag2, labels)\) to broadcast to all processes to confirm its reception of \((m, tag)\) (Lines 16-18).

- If \(p_i\) has received a \((m, tag)\) before (i.e., if this \((m, tag)\) has already existed in \(MSG_i\) and its ACK message \((m, tag, tag2)\) also exists in \(MY\_ACK_i\), it just re-read the label information from the failure detector and re-broadcast the corresponding acknowledge

---

<table>
<thead>
<tr>
<th>Initialization</th>
<th>23 (\text{end if})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (\text{sets MSG}_i, MY_ACK_i, URB_DELIVERED_i)</td>
<td>24 if ((ACK, m, tag, tag2, -)) is received for the first time then</td>
</tr>
<tr>
<td>2 empty</td>
<td>25 (\text{for each label } \in \text{labels}_i, \text{do})</td>
</tr>
<tr>
<td>3 activate Task 1</td>
<td>26 (\text{label_counter}<em>{i}(\text{[m, tag], label}) \leftarrow \text{label_counter}</em>{i}(\text{[m, tag], label}) + 1)</td>
</tr>
<tr>
<td>4 When URB_broadcast(_i(m)) is executed</td>
<td>27 end for</td>
</tr>
<tr>
<td>5 (\text{tag } \leftarrow \text{random}_i())</td>
<td>28 else</td>
</tr>
<tr>
<td>6 insert ((m, tag)) into (MSG_i)</td>
<td>29 (\text{for each label } \in \text{labels}_i, \text{associated with})</td>
</tr>
<tr>
<td>7 When receive(_{(MSG, m, tag)}) is executed</td>
<td>30 ((m, tag, tag2)) do</td>
</tr>
<tr>
<td>8 if ((m, tag)) is not in (MSG_i) then</td>
<td>31 (\text{label_counter}<em>{i}(\text{[m, tag], label}) \leftarrow \text{label_counter}</em>{i}(\text{[m, tag], label}) + 1)</td>
</tr>
<tr>
<td>9 insert ((m, tag)) into (MSG_i)</td>
<td>32 end for</td>
</tr>
<tr>
<td>10 (\text{end if})</td>
<td>33 if (\exists (m, tag) \in URB_DELIVERED_i) &amp; (\exists (\text{label, number}) \in a_theta_i;)</td>
</tr>
<tr>
<td>11 if ((m, tag, tag2)) is in (MY_ACK_i) then</td>
<td>34 (\text{label_counter}_{i}(\text{[m, tag, label])} \geq \text{number}) then</td>
</tr>
<tr>
<td>12 (\text{labels}_i \leftarrow {\text{label}</td>
<td>{\text{label, -} \in a_theta_i} })</td>
</tr>
<tr>
<td>13 (\text{broadcast}_{i}(ACK, m, tag, tag2, labels))</td>
<td>36 (URB_deliver(_i(m)))</td>
</tr>
<tr>
<td>14 else</td>
<td>37 (\text{repeat forever})</td>
</tr>
<tr>
<td>15 (\text{tag2 } \leftarrow \text{random}_i())</td>
<td>38 (\text{for every message } (m, tag) \in MSG, \text{do})</td>
</tr>
<tr>
<td>16 insert ((m, tag, tag2)) into (MY_ACK_i)</td>
<td>39 (\text{broadcast}_{i}(MSG, m, tag))</td>
</tr>
<tr>
<td>17 (\text{labels}_i \leftarrow {\text{label}</td>
<td>{\text{label, -} \in a_theta_i}})</td>
</tr>
<tr>
<td>18 (\text{broadcast}_{i}(ACK, m, tag, tag2, labels))</td>
<td>41 (\text{end repeat})</td>
</tr>
<tr>
<td>19 (\text{end if})</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.7 The algorithm of URB in \(AAS\_F_n, [A\Theta]\) (code of \(p_i\))
message \((ACK,m,tag,tag2,label_i)\) to all processes in order to confirm its reception of \((m,tag)\) to overcome the fair lossy channel (Lines 11-13).

When \(p_i\) receives an acknowledge message \((ACK,m,tag,tag2,label_j)\) from \(p_j\), if the first time receives this ACK, it inserts \((m,tag,tag2)\) into the set that stores all received ACKs and allocates an array \(rec\_num[(m,tag),LABEL]\). Then, it counts each label that existed in the variable \(label_j\). If it is not the first time receives this ACK, but there exists new labels in the variable \(label_j\), \(p_i\) just count the number of new labels.

When the \(rec\_num[(m,tag),x]\) of one label \(x\) equal or greater than \(y\) (the number of label \(x\) outputted by \(A\Theta\)). Due to the accuracy property of \(A\Theta\) that there is at least one correct process has known this label \(x\), meaning at least one correct process has received this message \(m\), \(p_i\) can \(URB\_deliver()\) this \(m\) if it has not done.

**Correctness proof of this algorithm**

**Theorem 3.9** Algorithm in Figure 3.7 guarantees the property of Uniform Reliable Broadcast.

**Proof:** The proof is similar to the Theorem 3.6. □

**Remark 3.4** With the help of failure detector \(A\Theta\), the Uniform Reliable Broadcast abstraction can be implemented in anonymous asynchronous distributed systems under the assumption of any crashed process. But this algorithm is still non-quiescent. A message that \(URB\_delivered\) by any process has to be broadcast forever by every correct process to overcome the message lost caused by fair lossy channels. This non-quiescent problem will be solved in the following sections.

### 3.4.6 Quiescent Fault-tolerant Broadcast Algorithms

Both Reliable Broadcast and Uniform Reliable Broadcast algorithms proposed above are non-quiescent. In this section, quiescent algorithms of fault-tolerant broadcast are given. Firstly, a new class of anonymous failure detector class of \(AP^*\) is defined and implemented. Then, the quiescent algorithms of reliable broadcast and uniform reliable broadcast are introduced.

As discussed above, the reason why algorithms in Figure 3.4 and Figure 3.7 are non-quiescent is that each correct process has to broadcast all \(RB\_delivered\) or \(URB\_delivered\) messages forever in order to overcome message lost caused by the fair lossy communication channels. Therefore, the intuitive idea to make these algorithms to be non-quiescent is to
stop this forever broadcast when a message has been $RB(URB)_{delivered}$ by all correct processes, i.e., delete messages that have been $RB(URB)_{delivered}$ by all correct processes from the set $MSG$.

Then, the following question comes over that how to confirm a message has been $RB(URB)_{delivered}$ by all correct processes in an anonymous system. In order to answer this question, we propose to combine two mechanisms: (1) each process broadcasts an acknowledgment message after it has delivered a message $m$, and then collects acknowledgment messages of $m$ from all processes (including itself); (2) a failure detector is needed to obtain information of all correct processes. With these two mechanisms, a process $p_i$ can delete a message $m$ from $MSG_i$ when it has received acknowledgments of $m$ from all correct processes.

It is necessary to mention that these two mechanisms must not break the anonymity of the system.

**Failure Detector Class of $AP^*$**

Failure detectors were proposed by T. D. Chandra and S. Toueg in 1991 [50], which is defined by several properties. It can be considered as a module that gives failure information (unreliable) of processes. Normally, this failure information is composed by the identifier of process in non-anonymous distributed systems. However, in anonymous distributed systems, processes have no identifiers. So, the key point to define and implement a failure detector in an anonymous distributed system is how to identify a process without breaking the anonymity of the system. We follow the same way as the definition of $A\Theta$ to define $AP^*$. The idea is to assign a random label to each process as a temporal identifier. This assignment does not break the anonymity of system, because this label is given in the failure detector layer, and no process knows the mapping relationship between a label and a process.

As mentioned before, a process $p_i$ can delete a message $m$ from its $MSG_i$ when it has received acknowledgment messages of this $m$ from all correct processes. So, the failure detector has to output the information of all correct processes. It means that this failure detector must have a strong completeness property that eventually correct processes do not trust any process that crashes\(^1\), and a strong accuracy property that a process cannot be trusted once it is crashed (may be need a little time). This failure detector is called as perfect anonymous failure detector ($AP^*$), which is the counterpart of perfect failure detector ($P$) in non-anonymous systems.

\(^1\)We use the complement of a suspicion to describe strong completeness.
Definition of $AP^*$

The anonymous perfect failure detector $AP^*$ provides each process $p_i$ with a read-only local variable $a_{p_i^*}$ that contains several pairs of ($label$, $number$), similar to failure detector $A\Theta$. For example, $a_p = \{(label_1, number_1), ..., (label_i, number_i), ..., (label_n, number_n)\}$ if there are $n$ processes in the system, where $label_i$ is a label of one process $p_i$ and $number_i$ is the number of correct processes who have known $label_i$.

- $AP^*$-completeness: There is a time after which variables $a_{p_i^*}$ only contain pairs of ($label$, $number$) associated to correct processes.

- $AP^*$-accuracy: If a process crashes, the label of this process and the corresponding number to the label will be permanently removed from variables $a_{p_i^*}$.

It is assumed that the number of each label is monotonically non-increasing. Eventually the number of pairs of ($label$, $number$) is equal to the number of correct processes.

Let us define $AP^*$ more formally:

$S(label) = \{ i \mid \exists \tau \in \mathbb{N}: (label, -) \in a_{p_i^*}^{\tau_i}\}$. $S(label)$ is the set of all processes that know label at time $\tau$.

- $AP^*$-completeness: $\exists \tau \in \mathbb{N}, \forall i \in Correct, \forall \tau' \geq \tau, \forall (label, number) \in a_{p_i^*}^{\tau'}$: $|S(label) \cap Correct| = number$.

- $AP^*$-accuracy: $\forall i, j \in \Pi, i \in Correct, j \in Faulty, \exists \tau, \forall \tau' \geq \tau: (label_j, number_j) \notin a_{p_i^*}^{\tau'}$.

Note that eventually the number of pairs ($label$, $number$) output is equal to the number of correct processes. Moreover, the assignment of labels does not break the anonymity of the system, because labels are assigned and counted in the failure detector implementation, and no process knows the mapping relationship between labels and processes neither in the Reliable Broadcast layer nor in the failure detector layer.

The Algorithm of $AP^*$

It is impossible to implement a failure detector of the class $P$ without the synchrony-related assumptions in non-anonymous asynchronous systems [103]. Moreover, the anonymous system model is weaker than non-anonymous system. This result can be extended to the anonymous asynchronous systems. In this subsection, $AP^*$ is implemented in anonymous synchronous system model where processes are connected with reliable communication channels. The system model $ASS_R[\theta]$ is a system like $AAS_R[\theta]$ but synchronous, which means that the maximum time to execute a step is bounded and known by every process.

Description of the algorithm:

Each process has three set and one array:
Initialization
1. set rec_i and output_i empty
2. label_i ← random_i()

Repeat every β time units
3. broadcast(label_i)
4. set timer_i to Δ
5. end repeat

When receive_i(label_j) from p_j is executed
6. if (label_j, −) is not in rec_i then
7. insert (label_j, 0) into rec_i
8. end if
9. ACK_i ← random_i()
10. broadcast(label_j, ACK_i)

When receive_i(label_j, ACK_i) from p_j is executed
11. if (label_j, −) is not in rec_i then
12. insert (label_j, 1) into rec_i
13. else
14. replace (label_j, v) by (label_j, v + 1) in rec_i
15. end if

When timer_i expires:
16. output_i ← rec_i
17. empty set rec_i

Fig. 3.8 The algorithm of failure detector AP* in ASS_R[0] (code of p_i)

- Set rec_i, initialize to be empty, records the received label and the corresponding number of this label.
- Set Output_i, initialize to be empty, is the output set of the failure detector.

Initially, a random label is assigned to each process (Line 3). Each process broadcasts, every β time, its label as a heartbeat and set its timer_i to Δ (Lines 4-7). It is assumed that β > Δ.

When process p_i receives a heartbeat (label_j) from process p_j, it will do as follows:
- If this label_j has not existed in set rec_i yet, p_i insert (label_j, 0) into rec_i. Then, p_i get a random number (tag of the ACK of this heartbeat) from the random function and broadcast (label_j, ACK_i)(Lines 9-13).

When process p_i receives an ACK of a heartbeat (label_j, ACK_i) from p_j:
- If this label_j does not exist in the set rec_i, then, insert (label_j, 1) into rec_i (Lines 15, 16). This 1 means the Heartbeat of label_j has been acknowledged by one process.
- If this label_j has already existed in rec_i, p_i add one time to the number of label_j in rec_i (Line 18).
When the timer expires, output the set \( rec_i \). Then set \( rec_i \) to be empty (Lines 20-22).

**Correctness proof:**

**Lemma 3.20** *(AP*-completeness) There is a time after which the output variable \( a_{p_i^*} \) only contains pairs of (label, number), where labels are of correct processes and numbers are the number of correct processes who have known each label.

**Proof:** This completeness property of \( AP^* \) follows from the observation that, if a process \( p_i \) crashes, it does not send its heartbeat any more, after \( \Delta \) time, a correct process \( p_j \) will receive none heartbeat from \( p_i \). Hence, \( p_j \) will not broadcast the ACK of \( p_i \)’s heartbeat anymore. Then after \( \Delta \) time, due to the synchrony of the channel, \( p_i \) will be considered as crashed and be exempted from the output of \( p_j \). Others correct process will follow the same way as \( p_j \)’s that they will not output the label of \( p_i \) and its count number of ACK anymore. We complete the proof.

**Lemma 3.21** *(AP*-accuracy) If a process crashes, the label and number of this process will be eventually and permanently deleted from the output variable \( a_{p_i^*} \).

**Proof:** There are two cases:

- If a process crashes at the beginning of a round, it does not send its label to any process. It is obvious that no correct process will output the label and number of this crashed process permanently.

- If a process crashes in a round, some correct process may output the label and number of this crashed process in such a round. Then, in the next round, the same as case 1, the label and number of this crashed process will be deleted from the output of correct processes permanently. This can be guaranteed by the fact that the round trip delay of a heartbeat and its ACK is bounded by \( \Delta \) and \( \beta \) is greater than \( \Delta \).

We complete the proof.

**Theorem 3.10** Algorithm in Figure 3.8 guarantees the property of failure detector \( AP^* \).

**Proof:** According to Lemma 3.20 and Lemma 3.21, it is trivial to see that Theorem 3.10 is correct.
Fault-tolerant Broadcast in Anonymous Distributed Systems

The reason that causes the algorithm of RB in Figure 3.4 to be non-quiescent is the permanent periodical broadcast of received messages in Task 1. Hence, in this section we address the design of a quiescent algorithm implementing Reliable Broadcast. The approach followed consists in eventually deleting every message from the set $MSG$. According to the properties of Reliable Broadcast, the periodical broadcast of Task 1 could be safely terminated when all the messages $RB_{delivered}$ by any correct process have been received by all correct processes. In other words, we could delete a message from the set $MSG$ when it has been received by all correct processes. Based on the previous, the design of a quiescent algorithm is reduced to the following two sub-problems: (i) determining the set of all correct processes in the system, and (ii) confirming that a message has been received by all processes in this set. With failure detector $AP^*$, the first sub-problem (determining the set of all correct processes in the system) has been solved. The second sub-problem (confirming that a message has been received by all correct processes) can be solved by making every process broadcast an “ACK” message when it receives a “MSG” message. Based on this, a quiescent Reliable Broadcast algorithm in $AAS_{F_n,t}[AP^*]$ is given in Figure 3.9.

Description of the algorithm

The algorithm runs as follows. Now every process $p_i$ manages four sets, initially empty: $MSG_i$, $RB_{DELIVERED_i}$, $MY_{ACK}_i$ (which records all tag_ack generated by $p_i$), and $ALL_{ACK}_i$ (which records all tag_ack received by $p_i$). Similarly to the algorithm of Figure 3.4, when $p_i$ calls $RB_{broadcast}(m)$ (line 4), its $random_i()$ generates a random tag for $m$ firstly (line 5). Then, $p_i$ inserts $(m, tag)$ into $MSG_i$ (line 6), so that $m$ will be broadcast periodically to all processes in Task 1 (lines 49-51).

When $receive_i(MSG, m, tag)$ is executed (line 7), $p_i$ inserts $(m, tag)$ into $MSG_i$ if this is the first reception of $m$ (lines 7-12). After that, $p_i$ inserts $(m, tag)$ into $RB_{DELIVERED_i}$ and generates a random tag_ack. Then, $p_i$ broadcasts a reception acknowledgment message of $(m, tag)$, which is composed of both tag_ack and label information (read from $a_{p_i}^*$). After that, $p_i$ delivers $m$ (lines 16-23). Otherwise, i.e., if an acknowledgment message of $(m, tag)$ is recorded in $MY_{ACK}_i$ (line 13), then it means that $m$ has already been delivered by $p_i$. In that case, $p_i$ just broadcasts the recorded acknowledgment message of $(m, tag)$, but with the updated label information from $a_{p_i}^*$ (lines 14-15).

When process $p_i$ receives an acknowledgment message $(ACK, m, tag, tag_{ack}, labels_j)$ from process $p_j$ (note that $p_j$ could be $p_i$ itself), there are three cases to consider:

- $p_i$ receives for the first time an acknowledgment message of $(m, tag)$ (by checking whether $(m, tag)$ is recorded or not in the set $ALL_{ACK}_i$), which also means that this
Initialization
sets MSG, RB_DELIVERED, MY_ACK, ALL_ACK, empty
activate Task 1

When RB_broadcast(m) is executed
m.tag ← random()
insert (m, tag) into MSG;

When receive(MSG, m, tag) is executed
if (m, tag) is not in MSG, then
insert (m, tag) into MSG;
end if
if (m, tag) is not in RB_DELIVERED,
insert (m, tag) into RB_DELIVERED;
end if
if receive(ACK, m, tag, tag_ack, labels) is executed
if (m, tag, tag_ack) is not in ALL_ACK, then
allocate array label_counter[(m, tag), −]
allocate array all_labels[(m, tag), −]
end if
if (m, tag, tag_ack) is not in ALL_ACK, then
insert (m, tag, tag_ack) into ALL_ACK;
end if
all_labels[(m, tag), tag_ack] ← labels;
for each label ∈ labels, do
label_counter[(m, tag), label] ← label_counter[(m, tag), label] + 1
end for
for each label in labels, but not in all_labels[(m, tag), tag_ack] do
all_labels[(m, tag), tag_ack] ← all_labels[(m, tag), tag_ack] ∪ {label}
label_counter[(m, tag), label] ← label_counter[(m, tag), label] + 1
end for
for each label in all_labels[(m, tag), tag_ack] but not in labels, do
all_labels[(m, tag), tag_ack] ← all_labels[(m, tag), tag_ack] \ label
delete label_counter[(m, tag), label]
end for
for each label in both all_labels[(m, tag), tag_ack] and labels, do
label_counter[(m, tag), label] ← label_counter[(m, tag), label] - 1
end for
end for
end if

Task 1:
repeat forever
for every message (m, tag) in MSG, do
broadcast(MSG, m, tag)
if each pair of (label, number) ∈ a_p^t : label_counter[(m, tag), label] = number ∧ all_labels[(m, tag), −] = {label
[(label, −) ∈ a_p^t ] then
if (m, tag) is in RB_DELIVERED, then
delete (m, tag) from MSG;
end if
end if
end for
end if
end for
end repeat

Fig. 3.9 Quiescent Algorithm of RB in AAS_Fn,t[AP^*] (code for p_t)
is the first ACK message from process $p_j$ (one tag_ack represents one process). In this case, $p_i$ allocates an array $\text{label\_counter}_i[(m, \text{tag}), -]$ (used to record the number of processes who have known each label received in this ACK message and related to $(m, \text{tag})$), and an array $\text{all\_labels}_i[(m, \text{tag}), -]$ (used to record every label in each ACK message related to $(m, \text{tag})$) (lines 25-28).

- $p_i$ receives an ACK message coming from a new process (by checking whether $(m, \text{tag}, \text{tag\_ack})$ is recorded or not in $\text{ALL\_ACK}_i$). (Observe that the previous case is naturally included in this case, but this case considers not only the very first ACK but later ACKs from others processes). In this case, $p_i$ first inserts $(m, \text{tag}, \text{tag\_ack})$ into $\text{ALL\_ACK}_i$ and $\text{labels}_j$ into $\text{all\_labels}_i[(m, \text{tag}), \text{tag\_ack}]$. After that, for each received label in $\text{labels}_j$, $p_i$ increases its count number by 1 (1 means that every label is known by the process from which $\text{tag\_ack}$ has been received) (lines 29-34).

- $p_i$ receives a repeated ACK message (with the same tag_ack) (due to the periodical broadcast of messages to cope with fair lossy channels). There are two mutually exclusive cases: 1) repeated ACK with “more” (new) label information (lines 36-39); 2) repeated ACK with “less” label information (due to the accuracy property of $\text{AP}^*$, that may need some time to delete a label corresponding to a crashed process) (lines 40-46). In case 1, for each new label, $p_i$ inserts it into $\text{all\_labels}_i[(m, \text{tag}), \text{tag\_ack}]$ and increases its count number by 1. In case 2, for each disappeared label, $p_i$ deletes it from $\text{all\_labels}_i[(m, \text{tag}), \text{tag\_ack}]$ and its corresponding label_counter. Then, $p_i$ decreases the count number of received labels by 1 (since it was not accurate due to the ACK message from a faulty process).

In Task 1, for each pair of $(\text{label}, \text{number})$ in $a_{p_i}^*$, if (1) the counted number of each label $\text{label\_counter}_i[(m, \text{tag}), \text{label}]$ is equal to the corresponding output number of $a_{p_i}^*$ (which means that $p_i$ has received number different ACKs (tag_ack) of $(m, \text{tag})$), and (2) the received labels related to $(m, \text{tag})$ $\text{all\_labels}_i[(m, \text{tag}), -]$ are equal to the output labels of $a_{p_i}^* \{\text{label} \mid (\text{label}, -) \in a_{p_i}^*\}$ (which means that the received ACKs are from correct processes) (line 52), together with the fact that $(m, \text{tag})$ has already been $\text{RB\_delivered}$, then $p_i$ deletes $(m, \text{tag})$ from the $\text{MSG}_i$ set (line 54).

**Lemma 3.22** The algorithm in Figure 3.9 is a quiescent implementation of the Reliable Broadcast communication abstraction in $\text{AAS}_F^*_n\{\text{AP}^*\}$.

**Proof:** The proofs of the Validity, Agreement and Uniform Integrity properties of Reliable Broadcast are straightforward. We will now prove the quiescence property of the algorithm of Figure 3.9.
An algorithm is said to be quiescent when eventually no process sends messages. In the algorithm of Figure 3.9, it is obvious that the broadcast of ACK messages (lines 15 and 21) is caused by the reception of MSG messages (line 7). Hence, we only need to show that the number of broadcasts of MSG messages is finite. Moreover, by nature a faulty process can only broadcast a finite number of times each MSG messages. Hence, the rest of the proof only focuses on showing that each correct process broadcasts a finite number of times each MSG message.

It is easy to see that the broadcast of MSG messages occur only in Task 1. Let us consider two processes $p$ (correct) and $q$, such that $p$ broadcasts $(MSG, m, tag)$ periodically by Task 1.

- If $q$ is correct, then eventually both $p$ and $q$ receive this MSG due to the fairness property of fair lossy communication channels. $p$ RB_delivers $m$ when it receives MSG for the first time. Also, by the algorithm $q$ broadcasts $(ACK, m, tag, tag\_ack, label_q)$ every time it receives MSG. By the fairness property of channels, $p$ will receive some of those ACK messages. According to lines 29-47, $p$ will count every $label$ in the received ACK from $q$, such that $label\_counter_p((m, tag), label_q) = 2$ and $label\_counter_p((m, tag), label_p) = 2$. From the properties of the failure detector $AP^*$, the output of $AP^*_p$ is composed of $label$ and number of correct processes, e.g., $[(label_q, 2), (label_p, 2)]$. Then, the condition of line 52 is satisfied, and thus process $p$ deletes $(m, tag)$ from $MSG_i$ and the repeated broadcast of the MSG message is stopped, which proves that this case is quiescent.

- If $q$ has crashed, then $p$ will only receive ACK from itself and together with the accuracy property of $AP^*_p$, the $label$ and corresponding number of $q$ will eventually and permanently be removed from the output of $AP^*_p$. Again, the condition of line 52 is satisfied, which proves that this case is quiescent too.

The previous reasoning completes the proof of the quiescence property of the algorithm of Figure 3.9.

**Quiescent Uniform Reliable Broadcast in $AAS\_F_{n,t}[A\Theta, AP^*]$**

In this subsection, a quiescent uniform reliable broadcast algorithm is given. This algorithm is implemented in anonymous asynchronous distributed system enriched with two failure detectors $A\Theta$ and $AP^*$. As discussed in the previous section, $A\Theta$ is used to take place of the assumption of a majority of correct processes.

**Description of the algorithm:**
Each process initializes its four sets: \( MSG_i \), \( URB\_DELIVERED_i \), \( MY\_ACK_i \), \( ALL\_ACK_i \) and activates the Task 1 (lines 1-3). We take a process \( p_i \) as an example to simplify the description. When \( p_i \) calls \( URB\_broadcast(m) \), it generates a random \( tag \) to this message \( m \) and inserts \((m, tag)\) into set \( MSG_i \) (lines 4-6). Then, this \( m \) is broadcast to all processes forever in the Task 1 in the form of \((MSG, m, tag)\) (lines 52-54).

When \( p_i \) receives a message \((MSG, m, tag)\), it first checks if this \((m, tag)\) has already existed in its \( MSG_i \). If not, then it checks if \((m, tag)\) has already been \( URB\_delivered \) (lines 8, 9). If not, \( p_i \) inserts this message to \( MSG_i \) (line 10). Otherwise, \( p_i \) overlook this reception.

Then, there are three cases as in the algorithm of Figure 3.5:

- If \( p_i \) receives \((MSG, m, tag)\) from itself for the first time (i.e., if this \((m, tag)\) has already existed in \( MSG_i \), but its ACK message \((m, tag, tag\_ack)\) does not exist in \( MY\_ACK_i \)). Then, \( p_i \) goes to execute the line 17 that generates a random \( tag\_ack \) to tag the acknowledgment message of \((m, tag)\). Then, \( p_i \) inserts this acknowledgment message \((m, tag, tag\_ack)\) into its sets \( MY\_ACK_i \), and reads the label information from its failure detector \( A\Theta_i \). Then, \( p_i \) broadcasts \((ACK, m, tag, tag\_ack, labels_i)\) to all processes to acknowledge the reception of \((m, tag)\) (lines 17-20).

- If \( p_i \) receives \((MSG, m, tag)\) from others process for the first time (i.e., if this \((m, tag)\) does not exist in \( MSG_i \) or \( URB\_DELIVERED_i \), neither its ACK message \((m, tag, tag\_ack)\) does not exist in \( MY\_ACK_i \)). It inserts this message into \( MSG_i \) (line 10). Then, \( p_i \) does the same as the case 1 (lines 17-20).

- If \( p_i \) has received this \((m, tag)\) already (i.e., if this \((m, tag)\) has already existed in \( MSG_i \) or \( URB\_DELIVERED_i \) and its ACK message \((m, tag, tag\_ack)\) also exists in \( MY\_ACK_i \)), it re-broadcasts the identical acknowledgment message but with the updated label information \((ACK, m, tag, tag\_ack, labels_i)\) to all processes in order to confirm the reception of \((m, tag)\) to overcome the message lost caused by the fair lossy communication channels (lines 13-15).

When \( p_i \) receives an acknowledgment message \((ACK, m, tag, tag\_ack, labels_j)\) from \( p_j \) (could be itself), there are three cases as follows:

- \( p_i \) receives the very first ACK message of \((m, tag)\), which also means this is the first time receives an ACK message with \( tag\_ack \) (by checking \((m, tag)\) exists in the set \( ALL\_ACK_i \) or not), which also means this is the first ACK message from one process (\( tag\_ack \) represents a process).
pᵢ allocates an array label_counterᵢ[(m, tag), −] (used to record the number of every label that received together with (m, tag)), and an array all_labelsᵢ[(m, tag), −] (used to record all label in each ACK message of (m, tag) (lines 23-25).

• pᵢ receives an ACK message coming from a new process (by checking (m, tag, tag_ack) exists in ALL_ACKᵢ or not). (Case 1 is naturally included in case 2, but case 2 considers not only the very first ACK but more later ACKs from others process).

  pᵢ firstly insert (m, tag, tag_ack) into the set ALL_ACKᵢ, and labelsⱼ into the array all_labelsᵢ[(m, tag), tag_ack]. After that, for each received label in labelsⱼ, pᵢ increases its count number by 1(lines 30-32) (1 means that every label is known by the process who generates this ACK with tag_ack).

• pᵢ receives a repeated ACK message (with the same tag_ack) (i.e., one process re-broadcast an ACK due to the fair lossy channel).

  There are two mutually exclusive cases: 1) repeated ACK with more (new) label information (lines 34-37); 2) repeated ACK with less label information (due to the completeness property of AΘ that it needs some time to delete a label of crashed process) (lines 38-44). In one instance of algorithm in Figure 3.9, only one of these two cases can happen. In case 1, for each new label, pᵢ inserts it into all_labelsⱼ[(m, tag), tag_ack] and increases its count number by 1. In case 2, for each disappeared label, pᵢ deletes it from all_labelsⱼ[(m, tag), tag_ack] and its corresponding label_counter. Then, decreases the count number of repeatedly received label by 1 (miscount this label by 1 due to the ACK from the crashed process).

  After counting the number of each label, if there exists one pair of (label, number) outputted by AΘᵢ satisfies the condition that the counted number of this label label_counterᵢ[(m, tag), label] is equal to the outputted number, then pᵢ checks this m has been URB_delivered or not. If not, pᵢ URB_deliver m for one time.

  In task 1, for each pair of (label, number) in the output of APᵢ⁺, if the condition that the counted number of each label label_counterᵢ[(m, tag), label] is equal to the corresponding number (means it has received number different ACKs(tag_ack) of (m, tag)) and the set of received label related to (m, tag) all_labelsⱼ[(m, tag), −] is equal to the outputted label set of APᵢ⁺ {label | (label, −) ∈ a_pᵢ⁺} (means the received ACKs (tag_ack) are from the correct processes) is satisfied (line 55), and together with the fact that (m, tag) has already been URB_delivered, then, pᵢ deletes (m, tag) from its MSGᵢ set (line 57).

Correctness proof:
Fig. 3.10 Quiescent algorithm of URB in AAS-Fn,(|A\Theta,Ap^n|) (code of pi)
Lemma 3.23 If a correct process broadcasts a message $m$, then it eventually deliver $m$. (Validity)

Proof: Let us consider a non-fault process $p_i$ broadcasts $m$. A unique random tag is assigned to this message $m$ (line 5), then inserts $(m, \text{tag})$ into the set MSG$_i$ to be broadcast a bounded but unknown times (until the condition of line 55 is satisfied) in Task 1 (lines 52-54). Together with the fairness property of fair lossy channel, all correct processes (include $p_i$) will receive this $m$ eventually.

Then, when a correct process receives $(\text{MSG}, m, \text{tag})$ for the first time, it generates a second unique tag$_{ack}$ to the corresponding acknowledgment message and broadcasts it to all processes. Due to the bounded but unknown times of broadcast$(\text{MSG}, m, \text{tag})$ in the Task 1 of $p_i$, each correct process receives it for a bounded but unknown times. Hence, each process broadcast an acknowledgment message for a bounded but unknown times too. The same reason of the fairness property of the communication channels, $p_i$ will receive all acknowledgment messages of $(m, \text{tag})$ from correct processes. Then, it is obvious that the condition of line 46 is satisfied, and $p_i$ delivers $m$. We complete the proof of Lemma 3.23.

Lemma 3.24 If some process deliver a message $m$, then all correct processes eventually deliver $m$. (Uniform Agreement)

Proof: To prove this Lemma, we consider the following two cases:
Case 1: A message $m$ is delivered by a correct process.

Suppose this correct process is $p_i$, then according to lines 52-54 of Task 1, $p_i$ will broadcast $m$ for a bounded but unknown times (until the condition of line 55 is satisfied) to all processes. With the fairness property of the channels, all correct processes will eventually receive $m$. Then, all correct process will do the same as $p_i$ to broadcast this $m$ a bounded but unknown times. Together with the Lemma 3.23, all correct processes eventually deliver this $m$.

Case 2: A message $m$ is delivered by a crashed process.

The condition of line 46 was satisfied before this crashed process deliver $m$. Due to the accuracy property of $A\Theta$, at least one correct process has received this $m$. Then, this correct process will broadcast $m$ for a bounded but unknown times (until the condition of line 55 is satisfied). Together with Lemma 1, it is obvious that all correct processes will deliver $m$.

Following case 1 and 2, we can see that Lemma 3.24 is correct.

Lemma 3.25 For every message $m$, every process delivers $m$ at most once, and only if $m$ was previously broadcast by sender$(m)$. (Uniform Integrity)
Proof: It is easy to see that any message \( m \) was previously broadcast by its sender, because each process only forwards messages it has received and the fair lossy channel does not create, duplicate, or garble messages.

To prove a message only be delivered for at most one time, let us observe that two kinds of tags exist in the system: one is used to label the message itself; one is used to label the acknowledgment of this message. The set \( MY_{ACK} \) is used to guarantee that each process broadcasts the identical acknowledgment message to the same \((m, tag)\) (line 18). The set \( URB_{DELIVERED} \) to record all messages that have delivered (line 48).

Even each message is broadcast for a bounded but unknown times (until the condition of line 55 is satisfied) and will be received by every correct process for a bounded but unknown times (lines 52-54), one message can not be modified or relabeled as a new message due to these tags and sets mentioned above. Moreover, every message is checked whether it has already existed in its set \( URB_{DELIVERED} \) (line 47) before \( URB_{deliver} \) it. With those mechanisms, it is certain that no message \( m \) will be delivered more than once. Hence, the proof is completed.

\[ \square \]

**Theorem 3.11** Algorithm in Figure 3.10 is a quiescent implementation of the Uniform Reliable Broadcast communication abstraction in \( AAS_{F,n,t}[A\Theta, AP^*] \).

Proof: From Lemma 3.23, 3.24 and 3.25, it is easy to see that Algorithm in Figure 3.10 is the implementation of the Uniform Reliable Broadcast. Then, it is only necessary to prove the algorithm in Figure 3.10 satisfies the quiescent property.

An algorithm is quiescent means that eventually no process sends or receives messages. In algorithm of Figure 3.10, it is obvious that the broadcast of \( ACK \) message is invoked by the reception of \( MSG \) message. Hence, the proof is reduced to show that the broadcast times of \( MSG \) message is finite. Moreover, a faulty process only broadcast a finite times of \( MSG \) message. Hence, the rest of this proof focus on that each correct process broadcast a finite times of \( MSG \) message.

It is easy to see that the broadcast of \( MSG \) only exist in Task 1. Let us consider two processes \( p \) (correct) and \( q \) that \( p \) broadcast \((MSG,m,tag)\) to \( q \) a bounded but unknown times \((p \text{ repeat broadcast } (MSG,m,tag) \text{ until the condition of line 55 is satisfied}).

- If \( q \) is correct, then eventually both \( p \) and \( q \) receive this \( MSG \) for a bounded but unknown times due to the fairness property of fair lossy communication channels, then \( p \) delivers \( m \) only once when the first reception of \( m \). Since \( q \) broadcast \((ACK,m,tag,label_{q})\) to \( p \) when each time it receives \( MSG \), \( q \) broadcast \( ACK \) to \( p \) for the same times as the reception times of \( MSG \). (\( p \) also can receive \( MSG \) from itself and broadcast its \( ACK \).
Here, we only take $q$ as an example.) By the fairness property of channels, $p$ receives a bounded but unknown times of $ACK$. According to lines 22-51, $p$ has to count every label existed in the received $ACK$ from $q$ and itself, as $label\_counter_p[(m, tag), label_q]=2$, $label\_counter_p[(m, tag), label_p]=2$. Together by the property of the failure detector $AP^*$, the output of $AP^*_p$ is composed by label and number of correct processes that is $[((label_q, 2), (label_p, 2))]$. Then, the condition of line 55 is satisfied that $(m, tag)$ is deleted from $MSG$ and $p$ stops the repeated broadcast of $(MSG, m, tag)$, which proves this case is quiescent.

- If $q$ is faulty. Then, $p$ only receives $ACK$ from itself and together with the accuracy property of $AP^*_p$, the label and corresponding number of $q$ will eventually and permanently removed from the output of $AP^*_p$. Hence, it is trivial that the condition of line 55 is satisfied, which proves this case is quiescent too.

Hence, according to the description mentioned above, we complete the proof. \hfill \Box

### 3.5 Related Work

Fault-tolerant broadcast abstraction is an important communication abstraction in distributed systems. It has several forms depending the degree of delivery guarantee. The weakest form is **reliable broadcast** (RB) with RB-broadcast() and RB-deliver() operations. RB was introduced in [113], which offers some degree of delivery guarantees. In short, RB is a broadcast service that requires that all non-crashed processes deliver the same set of messages, and that all messages sent by non-crashed processes must be delivered by all non-crashed processes. That situation can cause some inconsistencies when a process crashes after RB-deliver a message. For example, if a process RB-deliver(m), and it crashes after that, it is possible that none non-crashed process never executes RB-deliver(m). To tackle it, there is a stronger abstraction called **uniform reliable broadcast** that was proposed by V. Hadzilacos and S. Toueg ([78], [79]), with URB-broadcast() and URB-deliver() operations. It guarantees that if a process (no matter crashed or non-crashed) delivers a message $m$, then all non-crashed processes must deliver this message $m$.

In the atomic broadcast problem, processes have to agree on a set of messages and a delivery order for those messages. Introduction and survey on algorithms addressing Atomic Broadcast can be found in ([58], [59]). [50] showed that consensus and Atomic Broadcast are equivalent in asynchronous crash prone systems, i.e., Atomic Broadcast can be reduced to consensus and vice versa.
Many papers in the literature present the broadcasting primitives analyzing how hardware and software can work in concert on scalable multi-processor and also distributed systems, and show how these primitives can be used as building blocks for more complex parallel operations ([11], [12], [13], [14], [15], [16], [30], [31], [32], [117]). In these papers can be found a number of illustrative examples and applications for broadcasting and also fault-tolerance.

To our knowledge, all works that study the fault-tolerant broadcast services rely on distributed systems where processes are distinguishable because each one of them has a unique identity ([25], [27], [34], [51], [59], [74], [80], [103]), and communication channels are reliable (if a process $p$ sends a message to process $q$, and $q$ is correct, then $q$ eventually receives $m$) or quasi-reliable (if a process $p$ sends a message to process $q$, and the two processes are correct, then $q$ eventually receives $m$) [111]. However, real channels are neither reliable nor quasi-reliable, most of them are unreliable (e.g., fair lossy). Fair lossy channels means that if a message is sent an arbitrary but finite number of times, there is no guarantee on its reception, it can lose an infinite number of messages [28]. Many works have been done to construct reliable channels over unreliable channels in eponymous systems ([28], [1]).

Nevertheless, we can find in the literature several works addressing the problem of counting the size of a network where processes are anonymous and the network topology constantly changes ([69], [92], [95]). In these works failures are limited to links rather than processes.

Anonymous processes are common in some practical distributed systems, such as sensor networks, where a unique identity is not always possible to be included in each device (due to, for example, small storage capacity, reduced computational power, or a huge number of elements to be identified) ([7], [9], [100]). Another practical issue where anonymous processes are used is related with privacy (for example, to hide the user identity in a system) [55].
Chapter 4

Consensus in Anonymous Distributed Systems

4.1 Introduction

Consensus problem is a fundamental paradigm for fault-tolerant distributed computing. It can be solved easily in synchronous distributed system without process fault. However, it is well known that consensus cannot be solved deterministically in an asynchronous systems when there exists only one crashed process. In order to circumvent this impossibility, a failure detector was proposed by T. D. Chandra and S. Toueg [50]. These results of consensus in eponymous distributed system are applicable to homonymous systems too. In this chapter, we show that the $A\Omega'$ failure detector class is strictly weaker than $A\Omega$ (i.e., $A\Omega'$ provides less information about process crashes than $A\Omega$). We also present in this chapter the first implementation of $A\Omega'$ (hence, we also show that $A\Omega'$ is implementable), and, finally, we include the first implementation of consensus in anonymous asynchronous systems augmented with $A\Omega'$ and where a majority of processes does not crash.

4.2 System Model

In this chapter, we consider an anonymous distributed system (AS) as follows: the system is a message-passing system formed by a finite set $\Pi = \{p_i\}_{i=1,\ldots,n}$ of $n$ processes fully interconnected by channels. Each process $p_i \in \Pi$ uses the primitive broadcast to send a message to every process $p_j \in \Pi$. This primitive, denoted by $\text{broadcast}(m)$, sends a copy of message $m$ through each channel.
Processes are executed by taking steps. A process crashes when it stops taking steps. We assume that crashes are permanent. We say that process \( p_i \) is correct in a run if it does not crash, and faulty if \( p_i \) crashes. We denote by \( \text{Correct} \) the set of correct processes, and by \( \text{Faulty} \) the set of faulty processes. We denote by \( f \) the maximum number of processes that may crash in a run. We consider that if some process \( p_i \) crashes while the primitive \( \text{broadcast}(m) \) is invoked by \( p_i \), a copy of the message \( m \) can be delivered to any unknown subset of processes (including the empty subset).

For analysis, we assume that time advances at discrete steps. We also assume a global clock whose values are the positive natural numbers, but processes cannot access it. We use the notation \( \tau \in \mathbb{N} \) to indicate an instant of time.

Processes are anonymous [40]. Then, processes have no identity, and there is no way to differentiate between any two processes of the system (i.e., processes have no identifier and execute the same code).

A failure detector \( \text{FD} \) is a distributed device with a local module \( \text{FD}_i \) for each process \( p_i \in \Pi \). A failure detector \( \text{FD} \) returns information related with faulty processes each time that a process \( p_i \) invokes its module \( \text{FD}_i \). The addition of a failure detector \( \text{FD} \) in a system \( S \) (denoted by \( S[\text{FD}] \)) allows to solve a certain problem \( P \) that it is impossible to overcome in \( S \) alone. According to the type and the quality of the information about crashed processes, several classes of failure detectors have been proposed ([108], [103]).

4.3 Failure Detector Class \( A\Omega' \)

In this section, the algorithm \( A_{A\Omega'} \) implements the failure detector class \( A\Omega' \) in anonymous partially synchronous systems. This algorithm has a nice property: communication efficiency. That is, in every run, there is a time after which only leader processes broadcast messages.

We introduce in this section the algorithm \( A_{A\Omega'} \) to implement the failure detector \( A\Omega' \) in anonymous partially synchronous systems. This algorithm has a nice property: communication-efficiency. That is, in every run, there is a time after which only leader processes broadcast messages.

4.3.1 Definition of \( A\Omega' \)

The \( A\Omega' \) [43] failure detector provides each process \( p_i \in \Pi \) with two output variables \( \text{leader}_i \) and \( \text{quantity}_i \). Let \( L \) (resp., \( NL \)) be the subset of correct processes such that eventually their variable \( \text{leader} = \text{true} \) (resp., \( \text{leader} = \text{false} \)) permanently. We say that a correct process \( p_i \) is an eventually leader process (for shorten, a leader) if \( p_i \in L \), and an eventually non-leader
4.3 Failure Detector Class $A\Omega'$

A failure detector of class $A\Omega'$ [43] satisfies that:

1. Every correct process is either an eventually leader process, or an eventually non-leader process.
2. There is at least one eventually leader process in the system.
3. There is a time after which every eventually leader process $p_i$ has $\text{quantity}_i = |L|$ permanently, being $L$ the set of eventually leader processes in the system.

More formally, the definition of $A\Omega'$ is the following. Let $\text{leader}_i^\tau$ and $\text{quantity}_i^\tau$ be the variables $\text{leader}_i$ and $\text{quantity}_i$ provided by $A\Omega'$ at time $\tau$. Let $L = \{p_i \in \text{Correct} : \exists \tau : \forall \tau' \geq \tau, \text{leader}_i^\tau = \text{true}\}$, and $NL = \{p_i \in \text{Correct} : \exists \tau : \forall \tau' \geq \tau, \text{leader}_i^\tau = \text{false}\}$. In each run $R$ of the system, any failure detector of class $A\Omega'$ must satisfy the following three properties:

1. $(L \cup NL = \text{Correct}) \land (L \cap NL = \emptyset)$.
2. $L \neq \emptyset$.
3. $\exists \tau : \forall \tau' \geq \tau, \forall p_i \in L, \text{quantity}_i^{\tau'} = |L|$.

Note that there is not a time after which a correct process $p_k \in NL$ must have in $\text{quantity}_k$ the number of leaders $|L|$ of the system.

4.3.2 $A\Omega'$ is Strictly Weaker than $A\Omega$

First of all, we remind the definition of failure detector class of $A\Omega$ [41]. Let us consider that each process $p_i \in \Pi$ has a boolean variable $l_i$. Every failure detector of class $A\Omega$ satisfies eventually that: (1) there is a correct process $p_l$ that has $l_l = \text{true}$ permanently, and (2) every correct process $p_j$ other than $p_l$ has $l_j = \text{false}$ permanently. More formally, $\exists \tau \in N, \exists p_l \in \text{Correct} : \forall \tau' \geq \tau, \forall p_j \in \text{Correct}, l_l^{\tau'} = \text{true}$ and $l_j^{\tau'} = \text{false}$.

A failure detector of class $A\Omega$ satisfies that eventually (1) there is a correct process $p_i$ that has $\text{leader}_i = \text{true}$ permanently, and (2) every correct process $p_j$ other than $p_i$ has $\text{leader}_j = \text{false}$ permanently.

A failure detector class $X$ is strictly weaker than class $Y$ in system $S$ if (a) there is an algorithm that emulates the output of a failure detector $D'$ of class $X$ in the system $S$
augmented with a failure detector $D$ of class $Y$ (denoted by $S[D]$), and (b) the opposite is not true (i.e., there is no algorithm that emulates the output of a failure detector $D'$ of class $Y$ in the system $S$ augmented with a failure detector $D$ of class $X$). Then, we now prove that $A\Omega'$ is strictly weaker than $A\Omega$ in the following two cases:

**Lemma 4.1** Class $A\Omega'$ can be obtained from $AS[A\Omega]$.

**Proof:** Let $D$ be any failure detector of class $A\Omega$. Let $D'$ be an emulated failure detector with the following algorithm. Each process $p_i$ sets $D'.quantity_i = 1$, and permanently updates $D'.leader_i$ with the value of $D.l_i$.

From definition of $A\Omega$, eventually a single correct process $p_l$ has $D.\text{leader}_l = \text{true}$ permanently, and every correct process $p_j$ other than $p_l$ has $D.\text{leader}_j = \text{false}$ permanently. Hence, $p_l$ belongs to $L$, and the rest of correct processes belong to $NL$ (Condition 1 of $A\Omega'$). Then, $|L| = 1$, and, hence, $L \neq \emptyset$ (Condition 2 of $A\Omega'$). Finally, process $p_l$ has $D'.quantity_l = |L| = 1$ permanently (Condition 3 of $A\Omega'$). Therefore, $D'$ is a failure detector of class $A\Omega'$.

**Lemma 4.2** Class $A\Omega$ can not be obtained from $AS[A\Omega']$.

**Proof:** Let $D$ be a failure detector of class $A\Omega'$ with a run $R$ where the following six points are preserved: (1) the number of processes is greater than one, $|\Pi| > 1$, (2) all processes are correct, $\text{Correct} = \Pi$, and all of them are leaders, $L = \text{Correct}$, (3) from the beginning of the run, $D.\text{leader}_i = \text{true}$ and $D.\text{quantity}_i = |\text{Correct}|$ permanently in each process $p_i$ (note that this is one of the possible outputs of $A\Omega'$ by previous points (1) and (2)), (4) all processes execute in $R$ the same deterministic code at the same speed in lock step, broadcasting each message $m$ at the same time, (5) the delay of $m$ is the same in every channel, and, hence, $m$ will be received by every process in the same step of the execution, (6) if two messages $m$ and $m'$ are received in the same step, both messages will be delivered in the same order in every process.

Let us assume, by the way of contradiction, that $A\Omega$ can be deterministically obtained from $A\Omega'$ in all runs. Then, we construct a run $R$ as described above. Then, because the six points of $R$ and because processes have no identity, there is no way to distinguish among all correct processes in $R$ deterministically, and it is impossible to break this symmetry. Thus, every process $p_i$ either outputs $D'.l_i = \text{true}$ or $D'.l_i = \text{false}$ in $R$. Therefore, it is impossible to output $D'.l_l = \text{true}$ in a single correct process $p_l$, and $D'.l_j = \text{false}$ in every correct process $p_j$ other than $p_l$ in all executions (which contradicts the properties of $A\Omega$). Hence, a failure detector $D'$ of class $A\Omega$ can not be obtained from $AS[D]$.

**Theorem 4.1** $A\Omega'$ is strictly weaker than $A\Omega$.

**Proof:** It derives directly from Lemma 4.1 and Lemma 4.2.
4.3.3 Anonymous Partially Synchronous System APSS

Let APSS be a system like AS but with the following particular features. Channels are eventually timely. A channel between processes \( p_i \) and \( p_j \) is *eventually timely* if there is an (unknown) stabilization time \( T_{st} \) after which if process \( p_i \) sends a message at time \( t \geq T_{st} \), this message is delivered without errors to \( p_j \) in a bounded time \( t' \leq t + \Delta \), being \( \Delta \) an unknown but finite time. Messages sent by \( p_i \) at time \( t'' < T_{st} \) (i.e., before the global stabilization time) can be lost or delivered to \( p_j \) after a finite time greater than \( t'' + \Delta \).

We consider that the number of processes that may crash in the system APSS is at most \( n - 1 \) (i.e., \( f \leq n - 1 \)).

Processes are *partially synchronous* in the sense that the time to execute a step by a process \( p_i \) is an unknown positive but bounded time.

4.3.4 The Algorithm of \( A_{\Omega'} \) in APSS

The implementation algorithm of \( A_{\Omega'} \) in the system APSS is given in Figure 4.1. In every run, \( A_{\Omega'} \) eventually elects a set of leaders among all correct processes of the system APSS. This algorithm has a nice property: communication-efficiency. That is, in every run, there is a time after which only leader processes broadcast messages.

The description of the algorithm \( A_{\Omega'} \) of Figure 4.1 is as follows: a correct process \( p_i \) is one of the leader processes if the condition of line 15 in Task 1 is ever satisfied, and hence, \( \text{leader}_i \) contains *true* forever. Note that this is so because after line 1 there is no line in Task 1 and 2 of Figure 4.1 where \( \text{leader}_i \) is set to *false* again.

In Task 1, each leader process \( p_i \) broadcasts heartbeat messages (\( HB, seq_i \)) permanently, being \( seq_i \) its number of sequence (lines 5-8). A process \( p_i \) waits a time \( \text{timeout}_i \) (line 9) after which it checks how many acknowledgments it has received (lines 10-16). If process \( p_i \) is a leader process, it stores in \( rec_i \) the set of messages (\( ACK_{HB}, s, s' \)) received with \( s \leq seq_i \leq s' \) (line 11). Note that \( rec_i \), when this line 11 is executed, can return messages that had been received before line 7 is executed. Hence, \( \text{quantity}_i \) has the number of these acknowledgments contained in \( rec_i \) (line 12). If process \( p_i \) is not a leader process, it saves in \( rec_i \) the set of new messages (\( ACK_{HB}, -, - \)) received since its latest execution of line 14. If it does not receive any acknowledgment message, then process \( p_i \) becomes a leader (line 15).

In Task 2, each leader process \( p_i \) uses the variable \( \text{next}_\text{ack}_i \) to know the next number of sequence \( s \) of the acknowledgment message (\( ACK_{HB}, s, - \)) that process \( p_i \) has to broadcast. Initially, \( \text{next}_\text{ack}_i \leftarrow 1 \) (line 2). When a leader process \( p_i \) receives a message (\( HB, s_k \)) not previously acknowledged (i.e., \( s_k \geq \text{next}_\text{ack}_i \)) (line 18), it broadcasts a message (\( ACK_{HB}, \text{next}_\text{ack}_i \), \( s_k \))
Init:
1 \( \text{timeout}_i \leftarrow 1; \text{leader}_i \leftarrow 0; \)
2 \( \text{next}_\text{ack}_i \leftarrow 1; \text{quantity}_i \leftarrow 1; \)
3 \( \text{start} \) Task 1 and Task 2.

Task 1:
4 while true do
5 if \( (\text{leader}_i) \) then
6 \( \text{seq}_i \leftarrow \text{seq}_i + 1 \)
7 \( \text{broadcast}(\text{HB}, \text{seq}_i) \)
8 end if
9 wait until \( \text{timeout}_i \) units
10 if \( (\text{leader}_i) \) then
11 let \( \text{rec}_i \) be the set of \((\text{ACK}_\text{HB}, s, s')\) received such that \( s \leq \text{seq}_i \leq s' \)
12 \( \text{quantity}_i \leftarrow |\text{rec}_i| \)
13 else
14 let \( \text{rec}_i \) be the set of new \((\text{ACK}_\text{HB}, -, -)\) received
15 if \( (\text{rec}_i = 0) \) then \( \text{leader}_i \leftarrow \text{true} \) end if
16 end if
17 end while

Task 2:
18 upon reception of message \((\text{HB}, s_k)\) such that \( (s_k \geq \text{next}_\text{ack}_i) \) do
19 if \( (\text{leader}_i) \) then
20 \( \text{broadcast}(\text{ACK}_\text{HB}, \text{next}_\text{ack}_i, s_k) \)
21 \( \text{next}_\text{ack}_i \leftarrow s_k + 1 \)
22 end if
23 upon reception of message \((\text{ACK}_\text{HB}, s_k, s'_k)\) such that \( (S_k < \text{seq}_i) \) do
24 if \( (\text{leader}_i) \) then \( \text{timeout}_i \leftarrow \text{timeout}_i + 1 \) end if

Fig. 4.1 The algorithm of \( A_{\Omega'} \) in the system \( \text{APSS} \) (code of \( p_i \))

\( \text{next}_\text{ack}_i, s_k \) which acknowledges (in only one message) all heartbeat messages with the number of sequence in the range \( [\text{next}_\text{ack}_i, s_k] \) (line 20).

A leader process \( p_i \) may broadcast heartbeat messages \((\text{HB}, \text{seq}_i)\) faster than the time that another leader process \( p_k \) broadcasts messages \((\text{ACK}_\text{HB}, s_k, s'_k)\) with \( s_k \leq \text{seq}_i \). In this case, process \( p_i \) will receive outdated acknowledgment messages, and \( \text{timeout}_i \) will be incremented in one unit (lines 23, 24). Then, leader process \( p_i \) will slow down its heartbeat broadcasting speed because it increases the time that it is waiting at line 9.

4.3.5 Correctness of \( A_{\Omega'} \) in \( \text{APSS} \)

We now present the formal proofs to show that \( A_{\Omega'} \) implements \( A_{\Omega'} \) in \( \text{APSS} \).

The following lemma shows that there is a time after which every correct process \( p_i \) has \( \text{leader}_i = x \) permanently. This value \( x \) is either \text{true} or \text{false}.

Lemma 4.3 For each run, \((L \cup NL = \text{Correct}) \land (L \cap NL = \emptyset)\).
4.3 Failure Detector Class $A\Omega'$

**Proof:** Let us consider, by contradiction, that there is a run with a correct process $p_i$ such that $p_i \notin L$ and $p_i \notin NL$. Then, by this hypothesis of contradiction, there is some correct process $p_i$ such that $leader_i$ is changing its boolean value infinitely often. However, process $p_i$ initially has $leader_i = false$ (line 1), and it only may change to $true$ once (when the condition of line 15 is satisfied). Note that there is no line in Task 1 and Task 2 in the algorithm where $leader_i$ can be set to $false$ again. Hence, we reach a contradiction. Therefore, every correct process $p_i$ either $p_i \in L$ or $p_i \in NL$, and, hence, $(L \cup NL = Correct) \land (L \cap NL = \emptyset)$. 

Let $T_F$ be the time when every faulty process $p_f$ has crashed, and all messages $(HB, s)$ and $(ACK_HB, s)$ broadcast by $p_f$ have already been delivered or lost.

We prove in the following lemma that at least one correct process $p_c$ eventually has $leader_c = true$ permanently.

**Lemma 4.4** For each run, $L \neq \emptyset$.

**Proof:** By contradiction, let us consider that there is a run such that $L = \emptyset$. In the algorithm, if process $p_i$ changes from $leader_i = false$ (line 1) to $leader_i = true$ (line 15), $leader_i$ will never change to $false$ again. So, if the hypothesis of contradiction holds, there is no process that broadcasts message $(HB, s)$ and $(ACK_HB, s)$ after $T_F$, because $leader = false$ in all correct processes (lines 5-8 and lines 19-22). Note that the maximum number of faulty processes in the system is $n - 1$ (i.e., $f \leq n - 1$). Then, after $T_F$, at least one correct process $p_c$ will execute $leader_c \leftarrow true$, because it has not received any message since its latest execution of line 14, and $rec_c$ is empty (line 14, 15). Therefore, we reach a contradiction because at least a correct process $p_c$ has $leader_c = true$ permanently, and hence, for each run, $L \neq \emptyset$. 

Let $it_i^s$ be the $s^{th}$ iteration of process $p_i$. This iteration is formed by all operations from line 4 to line 17 of Task 1 that executed by process $p_i$ for the $s^{th}$ time.

We show in the following lemma that eventually each leader process $p_i$ has in $rec_i$, when it executes line 11, one (and only one) message $(ACK_HB, s, s')$ with $s \leq seq_i \leq s'$ of every leader process $p_j$.

**Lemma 4.5** In each run, given process $p_i \in L$ and $p_j \in L$, there is an iteration $it_i^{s_{a}}$ such that $\forall s_{b} \geq s_{a}$ process $p_j$ has in $rec_i$ exactly one message $(ACK_HB, s, s')$ with $s \leq s_{b} \leq s'$ of process $p_j$ when process $p_i$ executes line 11 at iteration $it_i^{s_{b}}$.

**Proof:** Note that, after executing $leader_i \leftarrow true$ of line 15, correct process $p_i \in L$ broadcasts messages $(HB, s_i)$ permanently, increasing in one unit the value of the sequence number $s_i$ at each iteration $it_i^{s_i}$. 
Let us define a time \( T_i \) such that \( T_i \geq T_{st} \), and process \( p_i \) and process \( p_j \) are already leaders. Then, leader process \( p_i \) will be broadcasting message \((HB, s_i)\) permanently at each iteration \( it^i_l \) with an increasing number of sequence \( s_j \), such that after time \( T_{st} \) we know that all of these heartbeat messages will be received by leader process \( p_j \in L \). So, we also know that process \( p_j \) after time \( T_{st} + \Delta \) will broadcast acknowledgment messages \((ACK\_HB, s_j, s_j')\) permanently with increasing values of \( s_j \) and \( s_j' \), being \( s_j \leq s_j' \). Note that process \( p_j \) broadcasts one (and only one) message \((ACK\_HB, s', s'')\) in response to all messages \((HB, s_i)\) received from all leaders, \( s' \leq s_i \leq s'' \) (lines 18-21).

Let us consider the following sequence of iteration numbers \( s_1 < s_2 \ldots < s_a \). Let \((ACK\_HB, s_1, -)\) be the first acknowledgment message broadcast by \( p_j \) after time \( T_1 \). Then, for each iteration \( it^i_l \), there is a message \((ACK\_HB, s, s')\) with \( s \leq s_2 \leq s' \) broadcast by process \( p_j \) and delivered at process \( p_i \) at most \( \Delta \) units of time after being broadcast. Note that \((ACK\_HB, s, s')\) with \( s \leq s_i \leq s' \) can be the same message for several consecutive iterations.

Note that if in an iteration \( it^i_l \), with \( s_x > s_1 \), when leader process \( p_i \) executes line 11, it has not received the message \((ACK\_HB, s, s')\) with \( s \leq s_x \leq s' \) from process \( p_j \), then, each time that this happens, \( timeout_i \) will be incremented when this message \((ACK\_HB, s, s')\) with \( s \leq s_x \leq s' \) is finally received (lines 23-24). This is so because \( seq_i \) will be greater than \( s_x \).

Let \( s_q \) be the iteration number where for the first time the value of \( timeout \) will be greater than time \( T_{reply_j} = 2\Delta + \phi_j \), being \( \Delta \) the maximum time to deliver a message from \( p_j \) to \( p_i \), and where \( \phi_j \) is the maximum time that process \( p_j \) takes to execute lines 18-22.

Now, let us assume, by contradiction, that there is an iteration \( it^i_l \), with \( s_b > s_a \), such that when leader process \( p_i \) executes line 11 at this iteration \( it^i_l \), it has not received the message \((ACK\_HB, s, s')\) with \( s \leq s_b \leq s' \) from process \( p_j \). Note that in this iteration process \( p_i \) broadcasts the message \((HB, s_b)\), and waits until \( timeout_i > T_{reply_j} \) because this time is never decreased in the algorithm. Then, when process \( p_i \) executes line 11 at this iteration \( it^i_l \), either: (a) will receive one message \((ACK\_HB, s, s')\) with \( s \leq s_b \leq s' \) from process \( p_j \), or (b) has already received one message \((ACK\_HB, s, s')\) with \( s \leq s_b \leq s' \) from process \( p_j \) in response to a faster leader.

Thus, for every iteration \( it^i_l \) with \( s_b \geq s_a \), exactly one message \((ACK\_HB, s, s')\) with \( s \leq s_b \leq s' \) from process \( p_j \) will be received by process \( p_i \) when it executes line 11 at \( it^i_l \). Hence, we reach a contradiction, and the claim of the lemma follows.

This theorem proves that \( A_{\Omega} \) is communication-efficient. Note that in the worst case all correct processes are in \( L \).

**Theorem 4.2** In the algorithm 4.1, there is a time after which only processes in \( L \) broadcast messages permanently.
4.4 Consensus with the Failure Detector $A\Omega'$

**Proof:** From Lemma 4.3 and definition of $T_F$, we can observe in the algorithm that eventually after $T_F$ only correct processes are alive and all broadcast and delivered messages belong to these correct processes. Then, if a correct process $p_i$ broadcasts a message $(HB, -)$ or $(ACK\_HB, -)$, it must have $leader_i = true$ (lines 5-8, and lines 19-22, respectively). So, if this case happens, it has already executed $leader_i \leftarrow true$ of line 15. Finally, note that if process $p_i$ changes from $leader_i = false$ (line 1) to $leader_i = true$ (line 15), this variable $leader_i$ will never change to $false$ again, and hence, $p_i$ is in $L$. Therefore, there is a time after which only processes in $L$ broadcasts messages permanently.

**Theorem 4.3** The algorithm of Figure 4.1 implements the failure detector $A\Omega'$ in APSS.

**Proof:** From Lemma 4.3 and Lemma 4.4, conditions 1 and 2 of $A\Omega'$ are preserved in each run. From Theorem 4.2 and Lemma 4.5, in each run, every process $p_i \in L$ eventually has $rec_i = L$ permanently when it executes line 11, and hence, $quantity_i = |L|$ (line 12). Thus, condition 3 of $A\Omega'$ is also preserved in each run. Therefore, the algorithm of Figure 4.1 implements the failure detector $A\Omega'$ in a system APSS.

### 4.4 Consensus with the Failure Detector $A\Omega'$

We introduce in this section the algorithm $A_{\text{cons}}$ to implement consensus in anonymous asynchronous systems augmented with the failure detector $A\Omega'$, and with a majority of correct processes (see Figure 4.2).

The consensus problem [50] specifies that all processes that take a decision have to decide the same value $v$, and this value $v$ has to be proposed by some process. More formally, the definition of consensus for anonymous systems is the following.

#### 4.4.1 Definition of Consensus

In each run, every process of the system proposes a value, and has to decide a value satisfying the following three properties:

1. **Validity:** Every decided value has to be proposed by some process of the system.

2. **Termination:** Every correct process of the system eventually has to decide a value.

3. **Agreement:** Every decided value has to be the same value.
4.4.2 Anonymous Asynchronous System $AAS_{Rn,t}$

Let $AAS$ be a system such as $AS$ but with the following particular features. Channels are reliable. A channel between processes $p_i$ and $p_j$ is reliable if every message sent by $p_i$ is delivered once to $p_j$ without errors in an unknown, positive and unbounded time. We consider that a majority of processes are correct in this system (i.e., $f < n/2$).

Each process $p_i$ initially has no information about any other different process $p_j$ of $\Pi (i \neq j)$ except that the size of the system is $n$ and $f < n/2$. In other words, in every run, process $p_i$ only knows that at least a majority of $n$ processes are correct, but it does not know who they are or the exact number of them.

As we have mentioned in the introduction of this Chapter 4, it is impossible to solve consensus in anonymous asynchronous systems. To avoid this result, failure detectors are included. We denote by $AAS[A\Omega']$ the anonymous asynchronous system defined in this section augmented with the failure detector $A\Omega'$.

4.4.3 The Algorithm Consensus $A_{cons}$ in $AAS_{Rn,t}[A\Omega']$

The algorithm of consensus in $AAS[A\Omega']$ is present in Figure 4.2. This algorithm is an adaptation of the leader-based consensus algorithm of [39] to the case in which multiple leaders coexist in the anonymous system. The changes between both algorithms are mainly focused in the phase $PH0$ where the failure detector is used. Every process $p_i$ uses the while sentence of Task 1 to execute asynchronous rounds permanently (lines 4-24). Each round is formed by three phases: $PH0$, $PH1$, and $PH2$. Process $p_i$ uses the variable $r_i$ to know the number of the round that it is executing. The variable $est_i$ contains the value proposed by $p_i$ in round $r_i$ to be decided. Note that initially $est_i$ contains the value $v_i$, that is, the proposal of process $p_i$ (line 1). The boolean variable $agree_i$ allows process $p_i$ to indicate whether it knows that a majority of processes has the same value in $est_i$ in round $r_i$. The boolean variable $l_i$ is used to know whether process $p_i$ is a leader process in round $r_i$.

In phase $PH0$, the goal is to reach a round $r$ after which every leader process $p_j$ has the same value $v$ in $est_j$ in each next round $r' \geq r$. To know if a process $p_i$ is a leader process in a round $x$, it saves in $l_i$ the boolean value of $leader_i$ returned by the failure detector $D$ of class $A\Omega'$ (line 5). If it is a leader, process $p_i$ broadcasts a message $(PH0, true, x, est_i)$ (line 6), and waits to receive a number $D.quantity_i$ of message $(PH0, true, x, -)$ (line 8). Note that this value $quantity_i$, returned by the failure detector $D$ of class $A\Omega'$, is eventually the number of all leader processes if process $p_i$ is a leader process (case 3 of $A\Omega'$). After that, a process $p_i$ sets in $est_i$ the minimal value of all received messages of $PH0$ (line 9-11), and broadcasts a message $(PH0, false, x, est_i)$ (line 12). This latter message allows non-leader processes to
4.4 Consensus with the Failure Detector $A\Omega'$

function $\text{propose}(v_i)$

Init:
1. $r_i \leftarrow 0$; $\text{est}_i \leftarrow v_i$
2. start Task 1 and Task 2

Task 1
3. while true do
4. $r_i \leftarrow r_i + 1$

% phase $\text{PH}0$
5. $l_i \leftarrow D.\text{leader}_i$
6. if ($l_i$) then broadcast($\text{PH}0, \text{true}, r_i, \text{est}_i$) end if
7. wait until

(l$_i \neq D.\text{leader}_i$) $\lor$ (l$_i \land D.\text{quantity}_i(\text{PH}0, \text{true}, r_i, \text{est}_i)$ received) $\lor$ ($\text{PH}0, \text{false}, r_i, \text{est}_i$) received
8. if (($\text{PH}0, \text{false}, r_i, \text{est}_i$) received) then
9. $\text{est}_i \leftarrow \min\{\text{est}_k: (\text{PH}0, \text{false}, r_i, \text{est}_k) \text{ received}\}$
10. end if
11. broadcast($\text{PH}0, \text{false}, r_i, \text{est}_i$)
% phase $\text{PH}1$
12. broadcast($\text{PH}1, r_i, \text{est}_i$)
13. wait until ($\text{PH}1, r_i, \text{est}_i$) received from a majority of processes
14. if (all ($\text{PH}1, r_i, \text{est}_i$) received: $\text{est}_i = \text{est}_i$) then
15. $\text{agree}_i \leftarrow \text{true}$ else $\text{agree}_i \leftarrow \text{false}$
16. end if
% phase $\text{PH}2$
17. broadcast($\text{PH}2, r_i, \text{est}_i, \text{agree}_i$)
18. wait until ($\text{PH}2, r_i, \text{est}_i, \text{agree}_i$) received from a majority of processes
19. if (($\text{PH}2, r_i, \text{est}, \text{true})$ received) then
20. $\text{est}_i \leftarrow \text{est}$
21. end if
22. if (all ($\text{PH}2, r_i, \text{est}, \text{true}$) received) then
23. broadcast($\text{DECIDE}, \text{est}_i$); return($\text{est}_i$)
24. end if
25. end while

Task 2
26. upon reception of ($\text{DECIDE}, v$) do
27. broadcast($\text{DECIDE}, v$); return($v$).

Fig. 4.2 The algorithm of consensus $A_{\text{cons}}$ in $\text{AAS}_{R_{n,t}}[\Omega']$ where is known that a majority of processes are correct (code of $p_i$)

finish of waiting in line 8 in round $x$. Note that there is an unknown time $t'$ after which the value $\text{leader}_i$ returned by $D$ stabilizes by definition of $A\Omega'$). Before this time $t'$, a process $p_i$ can believe that it is a leader in round $x$ because $l_i = \text{true}$, but $D.\text{leader}$ may change to $\text{false}$ in the same round $x$. In order to avoid being blocked forever in the wait sentence of line 7, the failure detector is checked permanently to know of the value of $l_i$ changes with respect to $D.\text{leader}_i$ while it is waiting in phase $\text{PH}0$ (line 8). Similarly, the value $D.\text{quantity}_i$ is also
checked permanently to eventually know the exact number of leader processes in this round $x$.

The goal of $PH1$ is that processes can check whether in a round $r$ each process $p_j$ of a majority of processes has the same value $v$ as the proposed value (i.e., $est_j = v$). To do so, each process $p_i$ broadcasts in round $x$ a message ($PH1, x, est_i$) (line 13), checking whether it receives the same value $est$ in all messages ($PH1, x, est$) from a majority of processes, and $est_i = est$ (line 15). If this happens, $agree_i ← true$, otherwise, $agree_i ← false$ (lines 15-17).

The goal of $PH2$ is that a process $p_i$ can decide a value $v$ in a round $r$. If this happens, $v$ has to be always the unique possible value to be decided by any other process $p_j$ in any of the next round $r' \geq r$. To do so, each process $p_i$ broadcasts, in a round $x$, a message ($PH2, x, est_i, agree_i$) and waits to receive a message ($PH2, x, −, −$) from a majority of processes (lines 18, 19). Note that, due to phase $PH1$, given any two received messages ($PH2, x, est_j, true$) and ($PH2, x, est_k, true$), the proposed value in this round $x$ has to be the same (i.e., $est_j = est_k$). Hence, if the fourth parameter of some received message is $true$, process $p_i$ establishes $est$ as the value to be decided in $x$ or in a next round (lines 20-22). On the other hand, if in all these messages the fourth parameter is $true$, process $p_i$ decides in this round $x$ the value $est$ of all received messages ($PH2, x, est, true$), and broadcasts a message ($DECIDE, est_i$) with its decision (lines 23-25).

4.4.4 Correctness of the algorithm $A_{cons}$ in $AAS\_R_{n,t}[A\Omega']$

We define a round $r$ as the set of sentences that every process $p_i$ executes while it has $r_i = r$.

**Lemma 4.6** Validity: For each run, every decided value has to be proposed by some process of the system.

**Proof:** Let us use induction on the number of rounds $r$ to show that when a process $p_i$ terminates a round in a run, it holds in its variable $est_i$ of process $p_i$ is initialized with its own proposed value $v_i$ (line 1). Induction hypothesis: assume that the claim is true for round $r=k$. Then, for every process $p_i$ that terminates round $k$, the value held in variable $est_i$, when $r_i = k$, was proposed by some process. Step $r=k +1$: we now show that the claim also holds in round $r = k +1$. For every process $p_i$ that finished round $k$, the variable $est_i$ can be changed in round $k+1$ with the proposed value $est$ broadcast in phase $PH0$ by some process (lines 9-11). After that, the variable $est_i$ can only be changed with proposed values $est$ broadcast by processes in phase $PH1$ (line 14) and $PH2$ (line 21). Then, the claim is also held in this Step $r = k +1$. Therefore, it is shown by induction that when a process $p_i$ terminates round $r$ it holds in its variable $est_i$ a value proposed by some process.
Thus, if a process $p_i$ terminates a round and decides $est_i = v$ when it executes line 24, this value $v$ was proposed by some process. On the other hand, if $p_i$ decides $v$ executing Task 2, this value $v$ was also proposed by some process because it is broadcast when line 24 is executed. So, for each run, every decided value has to be proposed by some process of the system.

**Observation 4.1** If some correct process does not wait forever at line 7 of a round $r$, then no other correct process will wait forever at line 7 of this round $r$.

**Proof:** If a correct process $p_i$ executes line 12, it broadcasts a message $(PH0, false, r, est_i)$ that makes the condition of line 8 true for every other correct process (recall that channels are reliable).

We say that a process $p_i$ changes its leading state if the value of $l_i$ is changed.

**Observation 4.2** If a correct process changes its leading state after executing line 5 of round $r$, then no correct process will wait forever at line 7 of this round $r$.

**Proof:** If a correct process changes its leading state after executing line 5 of round $r$, then the condition of line 8 becomes true and it unblocks. Then, from Observation 4.1, every other correct process stops waiting at line 7 of this round.

**Observation 4.3** If every correct process waits forever at line 7 of a round $r$, then the condition of line 8 evaluates to false forever for every correct process.

**Proof:** Otherwise, some correct process would stop waiting and, according to Observation 4.1, every correct process would stop waiting at line 7 of round $r$.

**Lemma 4.7** No correct process waits forever at line 7.

**Proof:** Let us assume, by the way of contradiction, that there is a correct process that waits forever at line 7 of a round $r$. Then, every other correct process waits forever too. Otherwise, Observation 4.1 would not hold. Besides, every correct process keeps its initial leading state. Otherwise, Observation 4.2 would not hold. Finally, since every other correct process waits forever, then from Observation 4.3, the condition of line 8 evaluates to false forever for every correct process eventually computes $D.quantity$ correctly. Since all the correct leader processes execute line 6, and they are leaders forever form Observation 4.2, then, eventually, at least one correct process will receive at least $D.quantity (PH0, false, r, −)$ messages (recall that the channels are reliable), what contradicts the initial assumption. completing the proof.
Lemma 4.8 Let $p_l$ be a leader process in run $R$. There is a round $r$ of run $R$ after which every process $p_i$ that terminates phase $PH0$ has $est_i = est_l$ at the end of phase $PH0$ of each round $r' \geq r$.

**Proof:** Let $t$ be the time in a run $R$ when (a) all faulty processes have crashed and their broadcast messages have already been delivered; (b) $D.leader$ does not change in any correct process anymore, and (c) $D.quantity$ does not change in any leader process anymore. Let $r$ be the largest round in run $R$ reached by any correct process at time $t$. Let us consider that this process is $p_i$. From Lemma 4.7, no process blocks in phase $PH1$ and $PH2$ of round $r$. Then, all leader processes eventually reach this round $r$ and broadcast $(PH0, true, r, est)$. Hence, we have two cases:

Case 1. Process $p_i$ is a leader process. Its variable $D.quantity_i$ has the total number of leader processes and it receives this number of messages $(PH0, true, r, est)$ (line 7). Then, it sets $est_i$ with the minimum value $est$ of all processes. Each leader process $p_l \neq p_i$ will also receive the same messages and will also set in its variable $est_l$ the same minimum value $est$. After that, process $p_i$ broadcasts $(PH0, false, r, est)$ (line 12). Therefore, variable $est$ of all leader processes have the same value.

Case 2. Process $p_i$ is a non-leader process. Each leader process, when finishes phase $PH0$, broadcasts $(PH0, false, r, est)$ with the minimum value $est$ of all leader processes (line 12). Hence, all messages $(PH0, false, r, est)$ received by process $p_i$ have the value $est$ of a same leader process.

Therefore, by the two previous cases, every process $p_i$ that terminates phase $PH0$ has $est_i$ with the same value of a leader process at the end of phase $PH0$ of this round $r$. Note that after phase $PH0$, the value in the variable $est$ does not change in the following two phases of the same round. Then, process $p_i$ keeps the same common value in $est_i$ in phase $PH1$ and $PH2$. Thus, in every round $r' \geq r$ of a run, every process $p_i$ that terminates phase $PH0$ of $r'$ has $est_i = est_l$ at the end of this phase $PH0$, being $p_l$ a leader process.

Lemma 4.9 Agreement: For each run, every decided value has to be the same value.

**Proof:** Let us suppose that a process $p_i$ decides a value $v$ in the round $r$ of a run, and a process $p_j$ decides a value $v'$ in round $r' \geq r$ of the same run. Then, this lemma is true if we show that $v = v'$.

Let us use induction on the number of round $r'$ to show this result. Let us assume that the base case of our induction is $r' = r$ (both processes $p_i$ and $p_j$ decide in the same round). If a process $p_i$ decides a value $v$ in this round $r$, it is because a majority of processes broadcast $(PH1, r, v)$ and $(PH2, r, v, true)$. Then, because two majority has at least one process in
common, \( v = v' \). Hence, the base case is satisfied. Induction hypothesis: Let us assume that the claim is true until round \( r' = r + k, l \geq 1 \). Then, if a process \( p_i \) decides a value \( v \) in round \( r \), every process \( p_j \) that decides until round \( r + k \) holds \( v \) in variable \( est_j \). Step \( k+1 \): We now show that the claims also holds in round \( r' = r + k + 1 \). Note that if a process decides a value \( v' \) in round \( r + k \), this process receives messages \((PH1, r+k, v')\) and \((PH2, r+k, v', true)\) of a majority of processes. Also note that if a process that does not decide in round \( r + k \) wants to terminate this round, it also has to receive messages \((PH2, r+k, -,-)\) of a majority of processes. Then, at least one message of this majority has to be \((PH2, r+k, v', true)\). By induction hypothesis, \( v' = v \). Hence, every process \( p_j \) that reaches round \( r + k + 1 \) holds in its variable \( est_j \) the value \( v \) in round \( r + k \). Clearly, if any process \( p_j \) decides in this round \( r + k + 1 \), the value only can be \( v \), hence, the claim is also satisfied for \( r' = r + k + 1 \). Therefore, the lemma is shown by induction.

Lemma 4.10 Termination: For each run, every correct process of the system eventually has to decide a value.

Proof: From Lemma 4.9, there is a round \( r \) in every run after which every process \( p_i \) that terminates phase \( PH0 \) has \( est_i = est_l, p_l \) is a leader process. Then, from Lemma 4.8 and because a majority of processes never crashes, all received messages in phase \( PH1 \) of round \( r \) are \((PH1, r, est_l)\) and its number is greater than \( n/2 \). Hence, process \( p_l \) does not block in phase \( PH1 \) and all received messages in phase \( PH2 \) of round \( r \) are \((PH2, r, est_l, true)\), and its number is also greater than \( n/2 \). So, every process \( p_l \) that terminates phase \( PH2 \) in round \( r \) can decides. Therefore, every correct process of the system decides.

Theorem 4.4 The algorithm described in Figure 4.2 solves the consensus problem in \text{AAS}[A\Omega'].

Proof: From Lemma 4.8, Lemma 4.9 and Lemma 4.10, validity, agreement and termination properties are satisfied in every run.

4.4.5 Analysis of rounds

A way to consider the costs of an algorithm for consensus in message-passing systems is to evaluate the number of rounds needed to decide a value. In [40] is shown that \( 2t + 1 \) is the lower bound on the number of rounds to achieve consensus (\( t \) is the maximum number of crashed processes). They can determine it exactly because they use the perfect failure detector for anonymous systems \( AP \). Differently of [40] because we do not use a perfect
failure detector, the maximum number of rounds in $A_{cons}$ for each run can not be bounded a priori. Therefore, we are going to analyze the extreme cases. Our algorithm $A_{cons}$ works in asynchronous consecutive rounds, such that each round is formed by three phases ($PH0$, $PH1$ and $PH2$).

The best case is when each process knows if it is a leader or a non-leader since the beginning of the run. That is, $(\forall p_i \in L, \forall p_k \in NL, \forall \tau) \implies (l^\tau_i = true$ and $l^\tau_k = false)$. Note that this happens if the failure detector $D$ of class $A\Omega'$ stabilizes since time $\tau = 0$ and returns true or false accurately. Hence, in this best case a process decides in the phase $PH2$ of the first round (line 24). The worst case for a correct process $p_i$ is if when the failure detector stabilizes, it is in the highest round $r'$ of all processes. In this case, $p_i$ has to wait until the rest of processes that form the majority reach its round $r'$ (line 7). Then, this process $p_i$ will decide in the phase $PH2$ of this round $r'$ (line 24).

4.5 Related Work

Consensus can be considered as a paradigm of agreement; processes propose an initial value and later have to decide by agreeing on one of the proposed values [50]. Consensus is deterministically solvable in synchronous systems. If we assume lower and upper bounds on process speeds and communication delays then consensus can be achieved despite the crash of any number of processes [93]. [73] proved that consensus can not deterministically be solved in an asynchronous system if a process may crash. This impossibility result, also called FLP, has had a great impact on this area, since it implies that there are also some other classical problems in distributed computing in crash prone asynchronous systems which are impossible to solve.

In order to circumvent the FLP result, several typical extensions have been proposed in non-homonymous systems as follows:

- Timing assumptions. Dwork, Lynch, and Stockmeyer [71]introduced the partial synchrony model. In this model, systems stabilizes to synchronous eventually, so, instead of using fixed timeouts, algorithms should implement adaptive timeouts. [70] considered 32 partially synchronous systems and showed that 4 out of those 32 models are minimal to solve consensus. [71] considered two partially synchronous models in which consensus is solvable, proved that consensus is solvable as long as $f < n/2$ and proposed some eventually synchronous consensus algorithms. [50] considered another partially synchronous model even weaker than the previous two ones of [71], and showed that consensus is solvable in that model as well.
• Failure detectors. A failure detector is an abstract module, available at every process, that informs in a possibly unreliable way about the failure information of processes in the system. They were proposed in [50]. With the information provided by failure detector, the FLP impossibility can be circumvented. Different failure detectors have been proposed depending on the problem and the implement system model.

• Randomization. Randomized models provide probabilities for some transitions. This means that instead of looking at a single worst case execution, one must consider a probability distribution over bad executions. If the termination requirement is weakened to require termination only with probability 1, the FLP argument no longer forbids consensus: non-terminating executions continue to exist, but they may collectively occur only with probability 0. There are two ways that randomness can be brought into the model. One is to assume that the model itself is randomized; instead of allowing arbitrary applicable operations to occur in each state, particular operations only occur with some probability [44]. Another one is to assume the randomness is located in the process themselves, processes are equipped with coin-flip operations that return random values according to some specified probability distributions [20].

Some consensus protocols combine aspects of protocols from different models. For example, a hybrid protocol of Aguilera and Toueg [4] solves consensus very quickly given a failure detector, but solves it eventually using randomization if the failure detector does not work.

In homonymous systems, there are also several research results. F. Bonnet and M. Raynal gave a systematic research about consensus problem in message passing homonymous systems. They proposed several failure detectors and also proved those relationships among them ([40], [39], [41]). Moreover, homonymous systems can be treated as a special case of homonymous systems. Several results in homonymous systems can be found in ([17], [61], [64], [60]). E. Ruppert [109] studied the homonymous solvability of consensus using shared objects of various types. C. Delporte et al. [67] provided several algorithms to emulate registers to solve consensus under different synchrony assumptions. They also investigated uniform consensus problem in homonymous systems with omission failures in [61]. In this paper, processes are divided in two cases: numerate processes (process can count identical message) and innumerate processes (process can not count identical message).
Chapter 5

Set Agreement in Homonymous Distributed Systems

5.1 Introduction

The $k$-set agreement problem [53] guarantees that from $n$ proposed values at most $k$ are decided. Two cases of this problem have received special attention: consensus (when $k = 1$), and set agreement (when $k = n - 1$). Traditionally, this problem is solved using a failure detector in asynchronous systems where processes may crash but not recover, where processes have different identities, and where all processes initially know the membership. In this chapter, we study the set agreement problem, and the weakest failure detector $\mathcal{L}$ used to solve it, in asynchronous message passing systems where processes may crash and recover, with homonyms (i.e., processes may have equal identities), and without a complete initial knowledge of the membership.

5.2 System Model

**Processes** The message passing system is formed by a set $\Pi$ of processes, such that the size $n$ of $\Pi$ is greater than 1. We use $id(i)$ to denote the identity of the process $p_i \in \Pi$.

**Homonymy** There could be homonymous processes [17], that is, different processes can have the same identity. More formally, let $ID$ be the set of different identities of all processes in $\Pi$. Then, $1 \leq |ID| \leq n$. So, in this system, $id(i)$ can be equal to $id(j)$ and $p_i$ be different of $p_j$ (we say in this cases that $p_i$ and $p_j$ are homonymous). Note that anonymous processes [41] are a particular case of homonymy where all processes have the same identity, that is, $id(i) = id(j)$, for all $p_i$ and $p_j$ of $\Pi$ (i.e., $|ID| = 1$).
Unknown knowledge of membership Every process $p_i \in \Pi$ initially knows its own identity $id(i)$, but $p_i$ does not know the identity of any subset of processes, or the size of any subset of $\Pi$, different of their trivial values. That is, process $p_i$ only knows initially that $id(i) \in ID$ and $|\Pi| > 1$.

Time Processes are asynchronous, and, for analysis, let us consider that time advances at discrete steps. We assume a global clock whose values are the positive natural numbers, but processes cannot access it.

Failures Our system uses basically the failure model of crash-recovery proposed in [3]. In this model processes can fail by crashing (i.e., stop taking steps), but crashed processes may have a recovery if they restart their execution (i.e., they may recover). A process is down while it is crashed, otherwise it is up. Let us define a run as the sequence of steps taken by processes while they are up. So, in every run, each process $p_i \in \Pi$ belongs to one of these five classes:

- **Permanently-up**: Process $p_i$ is always alive, i.e., $p_i$ never crashes.

- **Eventually-up**: Process $p_i$ crashes and recovers repeatedly a finite number of times (at least once), but eventually $p_i$, after a recovery, never crashes again, remaining alive forever.

- **Permanently-down**: Process $p_i$ is alive until it crashes, and it never recovers again.

- **Eventually-down**: Process $p_i$ crashes and recovers repeatedly a finite number of times (at least once), but eventually $p_i$, after a crash, never recovers again, remaining crashed forever.

- **Unstable**: Process $p_i$ crashes and recovers repeatedly an infinite number of times.

In a run, a permanently-down, eventually-down or unstable process is said to be incorrect. On the other hand, a permanently-up or eventually-up process in a run is said to be correct. The set of incorrect processes in a run is denoted by $Incorrect \subseteq \Pi$. The set of correct processes in a run is denoted by $Correct \subseteq \Pi$. Hence, $Incorrect \cup Correct = \Pi$.

Unless otherwise is said, we will assume that there is no limitation in the number of correct (or incorrect) processes in each run, that is, $t = n$ (being $t$ the maximum number of different processes that can crash and recover).

Features and use of the network The processes can invoke the primitive $broadcast(m)$ to send a message $m$ to all processes of the system (except itself). This communication primitive is modeled in the following way. The network is assumed to have a directed channel from process $p_i$ to process $p_j$ for each pair of processes $p_i, p_j \in \Pi \ (i \neq j)$. Then, $broadcast(m)$
invoked at process $p_i$ sends one copy of message $m$ along the channel from $p_i$ to $p_j$, for each $p_j \neq i \in \Pi$. If a process crashes while broadcasting a message, the message is received by an arbitrary subset of processes.

Unless otherwise is said, channels are asynchronous and fair-lossy [3]. A channel is fair-lossy if it can lose messages, but if a process $p_i$ sends a message $m$ permanently (i.e., an infinite number of times) to a correct process $p_j$, process $p_j$ receives $m$ permanently (i.e., infinitely often). A fair-lossy channel [3] does not duplicate or corrupt messages permanently, nor generates spurious messages.

**Process status after recovery** Following the same model of [3], when a process $p_i$ recovers, it has lost all values stored in its variables previously to crash, and it has also lost all previous received messages. A special case are *stable storage variables*. All values stored in this type of variables will remain available after a crash and recovery. Note that stable storage variables have their cost (in terms of operations latencies), and the algorithms have to reduce their use as far as possible.

Unless otherwise is stated, we consider, like in [3], that when a process $p_i$ crashes executing an algorithm $A$, if process $p_i$ recovers, it knows this fact, that is, $p_i$ starts executing from a established line of $A$ different of line 1.

**Nomenclature** The asynchronous system with homonymy and with unknown membership previously defined in this section is notated by $HAS_f[\emptyset, \emptyset, n]$. We denote by $HAS_f[X,Y,t]$ the system $HAS_f[\emptyset, \emptyset, n]$ augmented with the failure detector $X$ ($\emptyset$ means no failure detector), and where all processes initially know the identities of processes of $Y$ ($\emptyset$ means unknown membership). The third parameter $t$ indicates the maximum number of different processes that can crash and recover ($n$ means that all processes can crash and recover). The sub-index $f$ in the notation is used to denote that links are fair-lossy. For example, $HAS_f[\mathcal{L}, \Pi, n]$ denotes the asynchronous system with homonymous processes and fair-lossy links, enriched with the failure detector $\mathcal{L}$, where all processes initially know the identity of the members of $\Pi$, and where all processes can crash and recover. The classical definition of asynchronous systems found in the literature could be denoted by $AS_f[\emptyset, \Pi, t]$. That is, an asynchronous system without homonymy, with reliable links (i.e., where each sent message is delivered to all alive processes without errors and only once), where at most $t$ processes can crash, and where all processes initially know the identity of the members of $\Pi$.

We will use $HAS$ to denote a homonymous asynchronous system where the parameters are not relevant. Similarly, we use $AS$ instead of $HAS$ to indicate that it is a classical system where each process has a different identity.
5.3 Definitions

First, we formalize the set agreement problem [53].

**Definition 5.1 (Set agreement).** In each run, every process of the system proposes a value, and has to decide a value satisfying the following three properties:

- **Validity:** Every decided value has to be proposed by some process of the system.

- **Termination:** Every correct process of the system eventually has to decide some value.

- **Agreement:** The number of different decided values can be at most \( n - 1 \).

It is easy to see that if \( t = n \) and there are no stable storage variables, if all processes crash before deciding, and they recover later, all proposed values will be lost forever. Then, the Validity Property cannot be preserved, and, hence, set agreement cannot be solved. Thus, any algorithm that implements set agreement needs to use stable storage variables.

As in [3], we consider that a process \( p_i \) proposes a value \( v \) when process \( p_i \) writes \( v \) into a predetermined stable storage variable. Similarly, a process \( p_i \) decides a value \( v \) when process \( p_i \) writes \( v \) into another predetermined stable storage variable. Hence, after a recovery, a process \( p_i \), reading these variables, can know easily if a value has already been proposed and/or decided.

The set agreement problem cannot be solved in asynchronous systems where any number of processes can crash and not recover ([42], [81], [110]). To circumvent this impossibility result, we use a failure detector [50].

The failure detector \( \mathcal{L} \) [66] was defined for asynchronous systems with the crash-stop failure model. We adapt here the definition of \( \mathcal{L} \) to asynchronous systems where processes can crash and recover. Let us consider that each process \( p_i \) has a local boolean variable \( \text{output}_i \). We denote by \( \text{output}_i^\tau \) this variable at time \( \tau \). Let us assume that the value in \( \text{output}_i \) is \( \text{false} \) while process \( p_i \) is crashed (i.e., \( \text{output}_i^\tau = \text{false} \), at all time \( \tau \) at which \( p_i \) is down).

In each run, a failure detector of class \( \mathcal{L} \) satisfies the following two properties:

1. Some process \( p_i \) always returns in its variable \( \text{output}_i \) the value \( \text{false} \), and
2. If \( p_i \) is the unique correct process, then there is a time after which \( p_i \) always returns in its variable \( \text{output}_i \) the value \( \text{true} \).

More formally, the definition of \( \mathcal{L} \) for crash-recovery systems is the following.
5.4 Implementing Set Agreement in the Crash-Recovery Model

**Definition 5.2** (Failure detector $\mathcal{L}$). For every process $p_i \in \Pi$ and run $R$, $\text{output}_i^\tau = \text{false}$ if process $p_i$ is down at time $\tau$ in run $R$. Furthermore, the variable $\text{output}_i$ of every process $p_i \in \Pi$ must satisfy in each run $R$:

1. $\exists p_i : \forall \tau, \text{output}_i^\tau = \text{false}$, and
2. $(\text{Correct} = \{p_i\}) \implies \exists \tau : \forall \tau' \geq \tau, \text{output}_{i}^{\tau'} = \text{true}$

To solve set agreement, we augment our asynchronous system $\text{HAS}_f[\emptyset, \emptyset, n]$ with the loneliness failure detector $\mathcal{L}$, which is the weakest failure detector to achieve set agreement in classical asynchronous message passing systems $\text{AS}$ with the crash-stop failure model $[$? $]$. As we said previously, we denote this system enhanced with $\mathcal{L}$ as $\text{HAS}_f[\mathcal{L}, \emptyset, n]$.

**5.4 Implementing Set Agreement in the Crash-Recovery Model**

In this section, we present the algorithm $A_{\text{set}}$ (see Figure 5.1) that implements set agreement in homonymous asynchronous systems with unknown membership and with the failure detector $\mathcal{L}$, that is, in $\text{HAS}_f[\mathcal{L}, \emptyset, n]$.

Differently from $A_{\text{set}}$, all algorithms presented in the literature to solve set agreement with $\mathcal{L}$ ($[$33$]$ and $[$66$]$) work in crash-stop asynchronous systems and need to know the system membership.

**5.4.1 Description of the algorithm $A_{\text{set}}$**

$A_{\text{set}}$ is the algorithm of Figure 5.1 executed in $\text{HAS}_f[\mathcal{L}, \emptyset, n]$ to solve set agreement. Let $id(i)$ be the identifier of process $p_i$. Note that the values of these process identifiers could be whatever that imposes an order that allows to compare them. Also note that several identifiers can be the same (homonymous processes).

Recall that $id(i)$ is the identifier of process $p_i$. These identifiers are totally ordered which allows to compare them. Also recall that several identifiers can be the same (homonymous processes).

A process $p_i$ proposes a value $v$ (that is, $\text{propose}_i(v)$ is invoked) by writing $v$ into a stable storage variable $\text{PROP}_i$. Similarly, a process $p_i$ decides a value $v$ (that is, $\text{decide}_i(v)$ is invoked) by writing $v$ into another stable storage variable $\text{DEC}_i$. We assume that both variables have the value $\bot$ before any invocation. If a process $p_i$ recovers, it can see easily if it has already proposed or decided a value (that is, if $\text{propose}_i(v)$ or $\text{decide}_i(v)$ were invoked) reading these stable storage variables and checking if their values are different from $\bot$. 
propose$_i(v)$: % by writing $v$ into $PROP_i$
1  $v_i \leftarrow v$;
2  start task 1

**task 1:**
3  end$_i \leftarrow false$;
4  repeat each $\eta$ time
5    % Phase 0
6    broadcast ($PH0,id(i),v_i$);
7    if ($PH0,id(k),v_k$) is received then
8      if (($id(k),v_k$) $\leq$ ($id(i),v_i$)) then
9        $v_i \leftarrow v_k$;
10       decide$_i(v_k)$; % by writing $v_k$ into $DEC_i$
11       end$_i \leftarrow true$
12    end if
13    else
14      % Phase 1
15      if ($PH1,v_k$) is received then
16        $v_i \leftarrow v_k$;
17        decide$_i(v_k)$; % by writing $v_k$ into $DEC_i$
18        end$_i \leftarrow true$
19      else
20        if ($L$.output$_i$=true) then % returned by $\mathcal{L}$
21          decide$_i(v_i)$; % by writing $v_i$ into $DEC_i$
22          end$_i \leftarrow true$
23        end if
24      end if
25    until end$_i$;
26  start task 2

**task 2:**
27  repeat forever each $\eta$ time
28  broadcast ($PH1,v_i$);
29  end repeat

**when process $p_i$ recovers:**
% by checking $PROP_i$
30  if (propose$_i()$ was invoked) then
31     % by checking $DEC_i$
32     if (decide$_i()$ was invoked) then
33       $v_i \leftarrow DEC_i$;
34       start task 2
35     else
36       $v_i \leftarrow PROP_i$;
37       start task 1
38     end if
39  end if

Fig. 5.1 The algorithm of set agreement $A_{set}$ in $HAS_f[\mathcal{L},0,n]$. 
5.4 Implementing Set Agreement in the Crash-Recovery Model

The variable $v_i$ is used by process $p_i$ to keep the current estimate of its decision value (lines 9, 16). This variable $v_i$ contains initially the value $v$ proposed by process $p_i$ when it invokes $propose_i(v)$ (line 1). In order to remember, in case of recovering, the changes in $v_i$ before crashing, a process $p_i$ uses the stable storage variables $PROP_i$ and $DEC_i$ (lines 33, 36).

$propose_i(v)$ starts task 1. This task is a loop that executes lines 6 - 25 each $\eta$ time until a decision is taken (and, hence, variable $end_i = true$).

Each process $p_i$ in phase 0 broadcasts a message ($PH0, id(i), v_i$) with a proposal $v_i$ (initially $v_i$ is $p_i$’s proposal $v$, line 1) to the rest of processes of the system. After that, process $p_i$ can decide a proposed value if a message ($PH0, id(k), v_k$) is received. This value $v_k$ is only decided if the condition $\langle id(k), v_k \rangle \leq \langle id(i), v_i \rangle$ happens. This condition is a shortcut for $\langle id(k) < id(i) \rangle \lor [(id(k) = id(i)) \land (v_k \leq v_i)]$. That is, process $p_i$ decides $v_k$ if process $p_k$ has a lower identifier or, if they have the same identifier, $v_k$ is less than or equal to $v_i$. When a process decides, it moves to phase 1. If process $p_i$ has not decided in phase 0, it can decide a value already decided by another process if a message ($PH1, v_k$) is received. If, after that, phase 1 process $p_i$ has not decided yet, it can decide its value $v_i$ if the failure detector $L$ returns true (i.e., $L.output_i = true$). Note that at most $n - 1$ processes can get true in this variable $output_i$ (from Condition 1 of Definition 5.2).

Finally, if process $p_i$ decided in phase 0, phase 1, or locally because $L.output_i = true$, the loop of lines 4 - 26 finishes, and task 2 starts. As links are not reliable (but fair-lossy) and processes may crash and recover, with task 2 process $p_i$ guarantees the propagation of any decided value $v_i$ to the rest of processes. This value is broadcast in a message ($PH1, v_i$). The propagation is preserved by repeating forever this broadcast invocation (lines 28 - 30).

If a process $p_i$ crashes and recovers while running the algorithm, it always executes, after the recovery, lines 31-39. If process $p_i$ proposed a value $v$ but it crashed before writing any decision value in $DEC_i$, then $p_i$ will get the proposed value from the stable storage variable $PROP_i$ (line 36). Otherwise, $v_i$ will obtain its decided value from stable storage variable $DEC_i$ (line 33). If it has already proposed and decided a value, process $p_i$ starts task 2 to propagate this decided value (line 34). If process $p_i$ has proposed a value but it has not decided yet, it starts task 1 to look for a value to decide (line 37).

5.4.2 Proofs of $A_{set}$ in $HAS_f[\mathcal{L}, \emptyset, n]$

Lemma 5.1 (Validity) For each run, if a process $p_i$ of the system $HAS_f[\mathcal{L}, \emptyset, n]$ decides a value $v'$, then $v'$ has to be proposed by some process of the system $HAS_f[\mathcal{L}, \emptyset, n]$. 
Proof: The variable $v_i$ has initially, when $p_i$ starts for the first time, the value $v$ proposed by process $p_i$ when it invokes $\text{propose}_i(v)$ (line 1). Note that if process $p_i$ recovers after proposing a value $v$ but before writing any value in $\text{DEC}_i$, then $v_i = v$ (line 36). Thus, $v_i = v$ is broadcast in $(PH0, v_i)$ messages permanently (line 6 of $p_i$). So, this value $v_i = v$ only changes if:

Case 1: $(PH0, id(k), v')$ is received from some process $p_k$ such that $\langle id(k), v' \rangle \leq \langle id(i), v \rangle$ (lines 7-12 of $p_i$). Then, $v_i = v'$ and $\text{DEC}_i = v'$, being $v'$ the initial value proposed by process $p_k$.

Case 2: $(PH1, v')$ is received (lines 15-18 of $p_i$). We have three sub-cases:

Case 2.1: $(PH1, v')$ was broadcast by some process $p_j$ after receiving $(PH0, id(k), v')$ of $p_k$ ($p_k \neq p_j$) such that $\langle id(k), v' \rangle \leq \langle id(j), v \rangle$ (lines 7-12 and task 2 of $p_j$). Then, $v_i = v'$ and $\text{DEC}_i = v'$, being $v'$ the initial value proposed by process $p_k$.

Case 2.2: $(PH1, v')$ was broadcast by some process $p_j$ after receiving $(PH1, v')$ of other process $p_x$ (lines 15-18 and task 2 of $p_j$). Note that this $(PH1, v')$ is broadcast, like in Case 2.1, when process $p_x$ receives $(PH0, id(k), v')$ of some process $p_k$ such that $\langle id(k), v' \rangle \leq \langle id(x), v \rangle$. Then, $v_i = v'$ and $\text{DEC}_i = v'$, being $v'$ the initial value proposed by process $p_k$.

Case 2.3: $(PH1, v')$ was broadcast by process $p_k$ when $\text{output}_k = true$ (lines 20-23 and task 2 of $p_k$). Then, $v_i = v'$ and $\text{DEC}_i = v'$, being $v'$ the initial value proposed by process $p_k$.

Therefore, for each run, if a process $p_i$ of the system decides a value $v'$, then $v'$ has to be decided by some process of the system. □

Lemma 5.2 (Agreement) For each run, the number of different decided values in the system $\text{HAS}_f[\mathcal{L}, \emptyset, n]$ is at most $n - 1$.

Proof: Let us suppose, by the way of contradiction, that there is a run $R$ such that the number of different decided values is $n$. From Lemma 5.1, each decided value in $R$ has to be one of the proposed values. Hence, if we find in this run $R$ a proposed value which is not decided, we reach a contradiction.

Note that if in run $R$ there are two processes $p_i$ and $p_j$ such that $p_i$ proposes $v_i$, and $p_i$ proposes $v_j$ being $v_i = v_j$, then the statement of this lemma is trivial. So, we consider that $v_i \neq v_j$, for all $p_i$ and $p_j$ of the system.

Let us denote by $G$ the set of processes that decide in this run $R$ not executing lines 20-23. Note that $G \neq \emptyset$ from Condition 1 of Definition 5.2. Also note that this implies that every process $p_j \notin G$ decides its own proposed value.

Let us assume that $p_i \in G$ is the process with the greatest pair $\langle id(i), v \rangle$ among processes in $G$. Let us also assume that $p_i$ proposes $v_i$. So, if contradiction holds, $v_i$ has to be decided
by $p_i$ or by another different process $p_j$. We now analyze both cases and we will see that it is impossible that some process decides this value $v_i$ in run $R$. Hence, we reach a contradiction.

**Case 1:** Process $p_i$ decides $v_i$. As $p_i$, by definition, has the greatest pair $\langle id(i), v_i \rangle$ among processes in $G$, it did not receive any $(PH1, v_i)$ message from any process in $G$. Due to the fact that every process $p_j \notin G$ decides its own proposed value $v_j$ (being $v_j \neq v_i$), process $p_i$ did not receive any $(PH1, v_i)$ message from any process $p_j$. Then, it is impossible that process $p_i$ decides its own proposed value $v_i$.

**Case 2:** Process $p_j$ decides $v_i$, being $j \neq i$. As every process $p_k \notin G$ decides its own proposed value $v_k$ (being $v_k \neq v_i$), then process $p_j \in G$. Hence, if process $p_j$ decides $v_i$, which is a different value of its own proposed value $v_j$, it is because $p_j$ receives a $(PH0, id(l), v_i)$ or $(PH1, v_i)$ message from some process $p_l \in G$. This is impossible because, by definition, $p_i$ has the greatest pair $\langle id(i), v_i \rangle$ among processes in $G$, and $\langle id(i), v_i \rangle \leq \langle id(x), v_x \rangle$ is always false for all $p_x \in G$ (line 8).

Therefore, we reach a contradiction, and, for each run, the number of different decided values is at most $n - 1$. □

**Lemma 5.3** (Termination) For each run, every process $p_i \in Correct$ of the system $HAS_f[\mathcal{L}, \emptyset, n]$ eventually decides some value.

**Proof:** Let us suppose, by the way of contradiction, that there is a run $R$ such that a correct process $p_i$ never decides. Hence, if process $p_i \in Correct$ never decides in run $R$ it is because lines 10, 17 and 21 are never executed.

Let us prove that this situation is impossible. If line 21 is never executed, then $\mathcal{L}.output_{p_i} = false$ permanently. If this is so, it is because there is at least another process $p_k$ that is correct (from Condition 2 of Definition 5.2). Note that $p_i$, after its last recovery (if any), will be permanently broadcasting $(PH0, id(l), v_i)$ messages, being $v_i$ the proposed value of $p_i$ (line 6 of $p_i$). Hence, if process $p_i$ never receives $(PH1, -)$ messages (lines 15-18 of $p_i$) it is because all processes $p_l$ (included $p_k$) that receive the messages of $p_i$ have a lower pair $\langle id(l), v_i \rangle$ than $\langle id(i), v_i \rangle$ (line 8 of $p_i$). Nevertheless, process $p_i$ will receive $(PH0, id(k), v_k)$ messages of $p_k$ because links are fair-lossy, and correct process $p_k$ also broadcasts $(PH0, id(k), v_k)$ messages permanently (line 6 of $p_k$). Then, $p_i$ will execute line 10 because $\langle id(k), v_k \rangle < \langle id(i), v_i \rangle$. Hence, process $p_i$ will decide $v_k$ in run $R$. Therefore, we reach a contradiction, and, for each run, every process $p_i \in Correct$ eventually decides some value. □

**Theorem 5.1** The algorithm of Figure 5.1 implements set agreement in the system $HAS_f[\mathcal{L}, \emptyset, n]$.

**Proof:** From Lemma 5.1, Lemma 5.2 and Lemma 5.3, the validity, agreement and termination properties (respectively) are satisfied in every run. Hence, the algorithm of Figure 5.1 solves set agreement in the system $HAS_f[\mathcal{L}, \emptyset, n]$. □
5.5 On the Implementability of the Failure Detector $\mathcal{L}$ in Crash-Recovery Model

In this section we prove that the failure detector $\mathcal{L}$ can not be implemented, even in a synchronous system where the membership is known, if up to $n$ different processes can crash and recover; that is, $\mathcal{L}$ is not realistic [62]. We also prove in this section that the failure detector $\mathcal{L}$ can not be implemented in a partially synchronous system, even if the membership is known and up to $n-1$ different processes can crash and recover.

Let $SS_r[\emptyset, \Pi, n]$ be a system like $AS_r[\emptyset, \Pi, n]$ but synchronous, that is, the maximum time to execute a step is bounded and known by every process, and the time to deliver a message is also known by all processes. Hence, $SS_r[\emptyset, \Pi, n]$ is a synchronous system where all processes have different identities, links are reliable, the membership is known, and the maximum number of processes that can crash and recover is $t = n$. Similarly, let $PSS_r[\emptyset, \Pi, n]$ be a system like $SS_r[\emptyset, \Pi, n]$ but partially synchronous [70], that is, the maximum time to execute a step by each process $p_i$ is bounded, but unknown by every process different of $p_i$, and the time to deliver a message is bounded but unknown.

**Lemma 5.4** For every run, if in $SS_r[\emptyset, \Pi, t]$ or $PSS_r[\emptyset, \Pi, t]$ when $t \geq n - 1$ a process $p_i \in \text{Correct}$ stops receiving messages from the rest of processes at some time $\tau$, there is a time $\tau' \geq \tau$ where $\text{output}_i^{\tau'} = \text{true}$. 

**Proof:**

Let us assume, by the way of contradiction, that there is a run $R$ where some correct process $p_i$ stops receiving messages from the rest of processes at some time $\tau$, but for all time $\tau' \geq \tau$ it has $\text{output}_i^{\tau'} = \text{false}$. 

Let us consider another run $R'$ behaving exactly like $R$ until time $\tau$, and at this time $\tau$ all alive processes crash permanently except $p_i$. From Condition 2 of Definition 5.2 of $\mathcal{L}$, there is a time $\tau'$ where $\text{output}_i = \text{true}$. Note that each process only knows that in a run the rest of processes can crash, but it does not know a priori how many processes will crash or who they will be. Then, $R$ and $R'$ are indistinguishable until time $\tau'$ for $p_i$, and, hence, there is a time $\tau'$ where $\text{output}_i = \text{true}$ in $R$, which is a contradiction. 

The following theorem shows that failure detector $\mathcal{L}$ can not be implemented in $SS_r[\emptyset, \Pi, n]$.

**Theorem 5.2** There is no algorithm $A$ that implements the failure detector $\mathcal{L}$ in every run of a system $SS_r[\emptyset, \Pi, n]$, even if there is not any unstable process.
5.6 Implementing Failure Detector $\mathcal{L}$ in the Crash-Recovery Model

**Proof:** Let us assume, by the way of contradiction, that there is an algorithm $A$ that implements the failure detector $\mathcal{L}$ in every run of a system $SS_r[\emptyset, \Pi, n]$, even if there is not any unstable process.

For simplicity, let us consider that $\Pi = \{p_1, p_2, \ldots, p_n\}$, and that all these $n$ processes of $\Pi$ are eventually-up (hence, correct). Let us construct a valid run $R$ of $A$ as follows. For each process $p_i$, at time $\tau_i$ all processes crash except process $p_i$. From lemma 5.4, there is a time $\tau'_i \geq \tau_i$ where $output_i = true$. Now, all crashed processes recover at this time $\tau'_i$. Let $\tau_1 = 0$, and $\tau'_i < \tau_{i+1}$, $i = 1, \ldots, n$. Finally, after time $\tau'_n$ all processes keep alive in $R$ (i.e., there is no unstable processes). Then, at time $\tau'_n$ all processes have had $output = true$ at some time, which violates Condition 1 of Definition 5.2. Hence, we reach a contradiction.

Therefore, there is no algorithm $A$ that implements the failure detector $\mathcal{L}$ in every run of a system $SS_r[\emptyset, \Pi, n]$, even if there is not any unstable process.

The following theorem shows that failure detector $\mathcal{L}$ cannot be implemented in $PSS_r[\emptyset, \Pi, n-1]$.

**Theorem 5.3** There is no algorithm $A$ that implements the failure detector $\mathcal{L}$ in every run of a system $PSS_r[\emptyset, \Pi, n-1]$, even if there is not any unstable process.

**Proof:** From Lemma 5.4, there is a time $\tau_i$ after which each process $p_i \in Correct$ sets $output_i = true$ if it stops receiving messages from the rest of processes. Let us consider that every process $p_i$ in a run $R$ is permanently-up (hence, correct) and takes a step after a time $\tau$ which is greater than the maximum time $\tau_i$, for every process $p_i \in \Pi$. Note that processes do not know a priori the time needed by other processes to take a step in run $R$, nor the number of other processes that are correct in $R$. Hence, there is a time $\tau' \geq \tau$ after which every process $p_i$ has $output_i = true$, which violates the Condition 1 of Definition 5.2 of $\mathcal{L}$.

Therefore, there is no algorithm $A$ that implements the failure detector $\mathcal{L}$ in every run of a system $PSS_r[\emptyset, \Pi, n-1]$, even if there is not any unstable process.

### 5.6 Implementing Failure Detector $\mathcal{L}$ in the Crash-Recovery Model

From Section 5.5, we know that the failure detector $\mathcal{L}$ cannot be implemented in a synchronous system when up to $t = n$ processes can crash and recover; that is, $\mathcal{L}$ is not realistic [62]. From Section 5.5, we also know that $\mathcal{L}$ cannot be implemented in a partially synchronous system when $t = n - 1$. Now, we enrich here the system with a property such that we can circumvent these impossibility results. This property reduces to $t = n - 1$ the
number of processes that can crash and recover in a synchronous system. Note that all algorithms found in the literature that implement the loneliness failure detector $\mathcal{L}$ ([33], [99]) work in systems where processes can crash but not recover, where up to $t = n - 1$ processes can crash, and where the membership is totally known. Therefore, we present in this section an implementation of $\mathcal{L}$ (denote it by $A_{\mathcal{L}}$) for a synchronous system with homonymous processes, a partial knowledge of the membership, and where up to $t = n - 1$ different processes can crash and recover.

5.6.1 System Model

Let $HSS$ be a system like $HAS$ but synchronous. By synchronous we mean that processes start their execution at the same time, the time to execute a step is bounded and known by every process, the time to deliver a message sent through a link is at most $\Delta$ units of time, and this time is also known by all processes. For simplicity, we consider that the local execution time is negligible with respect to $\Delta$ (i.e., the execution time of a line of the algorithm is zero).

5.6.2 Algorithm $A_{\mathcal{L}}$

We show in this section that the algorithm $A_{\mathcal{L}}$ of Figure 5.2 implements the failure detector $\mathcal{L}$ in $HSS, [0, Y, n - 1]$ when $|Y| \geq 2$ and there are two processes of $Y$ that have different and known identities.

For each process $p_i$, $output_i$ is initially false (line 3). Process $p_i$ uses the boolean value of the stable storage variable $restarted_i$ to communicate to the other processes if it has ever crashed (initially the value of this variable is false, line 1). If process $p_i$ recovers, it will execute lines 19-20, and $restarted_i$ will be true (line 19). By definition of the system $HSS$ used to execute $A_{\mathcal{L}}$, process $p_i$ knows at least two processes’ identifiers with different values. These two known identifiers of $Y$ with different value are $IDENT_1$ and $IDENT_2$ in Figure 5.2. Then, each process $p_i$ whose identifier is neither $IDENT_1$ nor $IDENT_2$ changes $output_i$ to true (lines 4-6). Repeatedly, each process $p_i$ broadcasts heartbeats with messages ($alive, restarted_i$) that arrive synchronously (at most $\Delta$ units of time later) to the rest of the processes of the system (line 8). Note that we select a value $\eta$ greater than $\Delta$ to allow that messages broadcast in line 8 arrive to processes on time in each iteration of line 9.

After $\Delta$ units of time, process $p_i$ analyzes the messages received ($rec_i$) to see if it has to set $output_i$ to true (lines 11-17). Note that once $output_i = true$, process $p_i$ never changes it to false again while it is running. Only if process $p_i$ crashes and recovers, line 3 is executed again and $output_i$ is false again, but $restarted_i$ will be true in this case. The variable $count_i$ counts the number of heartbeats received by $p_i$ from processes that are up, and that have
never crashed (lines 12-14). If this number of messages is 0, then $p_i$ sets $output_i = true$ (lines 15-17).

Note that in all algorithms in the literature that implement set agreement with $\mathcal{L}$ (our algorithm $A_{set}$ included), the performance is improved if processes obtain $true$ from $\mathcal{L}$ as soon as possible. This happens because a process of set agreement can decide locally (without waiting to receive any message) if $true$ is returned by $\mathcal{L}$. For that reason, our algorithm $A_{\mathcal{L}}$ with a partial knowledge of the membership immediately sets $output = true$ permanently in $n-2$ processes (lines 4-6 of Figure 5.2).

```plaintext
init:
1 restarted, ← false;
2 start task 1

task 1:
3 output, ← false;
4 if ((id(i) ≠ IDENT1) ∧ (id(i) ≠ IDENT2)) then
5 output, ← true
6 end if
7 repeat forever each η time
8 broadcast (alive, restarted,);
9 wait Δ time;
10 let rec, be the set of (alive, restarted) messages received;
11 count, ← 0;
12 for_each ((alive, restarted) ∈ rec, such that restarted = false) do
13 count, ← count, + 1
14 end for_each
15 if (count, = 0) then
16 output, ← true
17 end if
18 end repeat

when process $p_i$ recovers:
19 restarted, ← true;
20 start task 1
```

Fig. 5.2 The algorithm of failure detector $\mathcal{L}$ (code of $p_i$)

**Lemma 5.5** For each run, there is a process $p_i$ of system $HSS_r[0,Y,n-1]$ where $|Y| ≥ 2$ and two processes of $Y$ have different identities, such that $output_i = false$ at all times of the run.

**Proof:** Note that in this system $HSS_r[0,Y,n-1]$, being $|Y| ≥ 2$, there are two different identifiers $id(j)$ and $id(k)$, being $p_j, p_k ∈ Y$, which are known by all processes of the system (denoted by IDENT1 and IDENT2 in the code of Figure 5.2). Then, processes $p_j$ and $p_k$ initially set $output_j = false$ and $output_k = false$ (lines 3-6).

Let $p_{up}$ be a permanently-up process. Note that in a system $HSS_r[0,Y,n-1]$ at most $n-1$ processes can crash, hence, there must be at least a process permanently-up. Note
that \( p_{up} \) always has \( \text{restarted}_{up} = \text{false} \) because it never crashes, and, therefore, it never executes line 19. So, due to fact that the system is synchronous with reliable links, each message \((\text{alive}, \text{restarted}_{up})\) with \( \text{restarted}_{up} = \text{false} \) sent by \( p_{up} \) arrives to every process (other than \( p_{up} \)) in at most \( \Delta \) units of time later. Hence, if \( p_{up} \neq p_k \) (or \( p_{up} \neq p_j \)), process \( p_k \) (or \( p_j \)) always receives on time these messages when it is up.

We have two cases to study:

**Case 1:** \( p_{up} = p_j \) or \( p_{up} = p_k \). Let us suppose, without loss of generality, that \( p_{up} = p_j \). Then, we have two sub-cases:

**Case 1.1:** process \( p_k \) is up. As at least a message \((\text{alive}, \text{restarted}_{up})\) with \( \text{restarted}_{up} = \text{false} \) is received, the variable \( \text{count}_k \) is \( \text{count}_k \geq 1 \), and \( p_k \) never executes line 16.

**Case 1.2:** process \( p_k \) is down. From Definition 5.2, \( \text{output}_k = \text{false} \) all the time that \( p_k \) is down.

**Case 2:** \( p_{up} \neq p_j \) and \( p_{up} \neq p_k \). Then, we also have two sub-cases:

**Case 2.1:** process \( p_k \) (or \( p_j \)) is up. As at least a message \((\text{alive}, \text{restarted}_{up})\) with \( \text{restarted}_{up} = \text{false} \) is received, variable \( \text{count}_k \) (or \( \text{count}_j \)) is greater than 1, and \( p_k \) (or \( p_j \)) never executes line 16.

**Case 2.2:** process \( p_k \) (or \( p_j \)) is down. From Definition 5.2, \( \text{output}_k = \text{false} \) (or \( \text{output}_j = \text{false} \)) all the time that process \( p_k \) (or \( p_j \)) is down.

Therefore, from two previous cases, we can observe that at least a process \((p_j \text{ or } p_k)\) always has its variable \( \text{output} = \text{false} \). Hence, for each run, there is a process \( p_i \) of system \( \text{HSS}_r[\emptyset, Y, n-1] \), where \( |Y| \geq 2 \) and two processes of \( Y \) have different identities, such that \( \text{output}_i = \text{false} \) at all times of the run.

**Lemma 5.6** For each run, if \( \text{Correct} = \{p_i\} \) in a system \( \text{HSS}_r[\emptyset, Y, n-1] \) where \( |Y| \geq 2 \) and two processes of \( Y \) have different identities, then there is a time after which \( \text{output}_i = \text{true} \) permanently.

**Proof:** If \( \text{Correct} = \{p_i\} \) then process \( p_j \in \text{Incorrect} \), for all \( p_j \) such that \( j \neq i \). So, each \( p_j \) is permanently-down, eventually-down, or unstable. Due to the fact that in the system up to \( n-1 \) processes can crash, then process \( p_i \) has to be permanently-up. Hence, there is a time \( \tau \) after which \( p_i; \) (a) stops receiving messages from \( p_j \) (if \( p_j \) is permanently-down or eventually-down), or (b) all received messages \((\text{alive}, \text{restarted}_j)\) from \( p_j \) will have \( \text{restarted}_j = \text{true} \) (if \( p_j \) is unstable or eventually-up). Hence, eventually \( \text{count}_i = 0 \) because there will not be messages \((\text{alive}, \text{restarted})\) received by process \( p_i \) with \( \text{restarted} = \text{false} \) (lines 12-14), and process \( p_i \) will execute line 16. Therefore, process \( p_i \) eventually has \( \text{output}_i = \text{true} \) permanently.
Theorem 5.4 The algorithm \( A_L \) implements the failure detector \( L \) in a system \( HSS_r[\emptyset, Y, n - 1] \) when \( |Y| \geq 2 \) and two processes of \( Y \) have different identities.

Proof: From Lemma 5.5 and Lemma 5.6, Conditions 1 and 2 of Definition 5.2 are satisfied in every run. Hence, the algorithm of Figure 5.2 implements the failure detector \( L \) in a system \( HSS_r[\emptyset, Y, n - 1] \) where \( |Y| \geq 2 \) and two processes of \( Y \) have different identities. 

5.7 Related Work

K-set agreement abstraction is considered as a generalization of consensus. It defines that a set of \( n \) processes can decide at most \( n - 1 \) from \( n \) proposed values.

Let \( t \) be an upper bound on the number of processes that may crash in a run, \( 1 \leq t < n \). Hence, \( t \) is a model parameter. If \( t < k \), k-set agreement can be trivially solved in both synchronous and asynchronous systems: \( k \) predetermined processes broadcast the values they propose and a process decides the first proposed value it receives. Hence, the interesting setting is when \( k \geq t \), i.e., when the number of values that can be decided is smaller or equal to the maximal number of processes that may crash in any run.

Although several solutions to this problem have been proposed for synchronous systems, ([42], [110]) showed that it is impossible to deterministically solve this problem in asynchronous systems when the number of faults is equal or bigger than \( k \); observe that if there is only one faulty process and \( k = 1 \) then the impossibility result of [73] can also be applied. As in consensus, several ways have been proposed to circumvent this impossibility result, such as randomization or failure detectors.

Failure detectors have been investigated to solve the k-set agreement problem since 2000 [97]. Random oracles to solve the k-set agreement problem have also been investigated [98]. Lower bounds to solve k-set agreement in asynchronous message passing systems enriched with limited accuracy failure detectors have been conjectured in [97]. The question of the weakest failure detector class for the k-set agreement problem (\( k > 1 \)) has been stated first in [106].

Consensus and failure detectors were presented in asynchronous systems where processes may crash and recover [3]. Besides processes that in a run do not crash (permanently-up) and processes that crash and stop forever (permanently-down), new classes of processes may appear in a run of a crash-recovery system: processes that crash and recover several times but after a time are always up (eventually-up), processes that crash and recover several times but after a time are always down (eventually-down), and processes that are permanently crashing and recovering (unstable). In these crash-recovery systems a process is said to be correct in
a run if it is either permanently-up or eventually-up. On the other hand, an incorrect process in a run is either a permanently-down, eventually-down or unstable process. In [3] is proven that consensus with the failure detector $\diamond P$ [50] is impossible to solve if the number of permanently-up processes in a run can be lesser or equal to the number of incorrect processes. There are in the literature several implementations of consensus and $\Omega$ for crash-recovery message passing systems ([3], [83], [94]).

Even though the initial knowledge of the membership is not always possible, different grades of knowledge are also possible. For example, $\Omega$ is implementable if each process initially only knows its own identity [84], or if each process also knows $n$ (i.e., the number of processes of the system) [18].

In [17] new classes of failure detectors are introduced to work in homonymous systems. In that paper consensus is also implemented with the counterparts of the weakest failure detectors in classical message passing systems with unique processes’ identities: $\Omega$ [49] when a majority of processes are correct (its counterpart is called $H\Omega$), and $\langle \Omega, \Sigma \rangle$ [63] when a majority of processes can crash (its counterpart is called $H\Omega, H\Sigma$).

Regarding set agreement in message passing systems, in the literature we find only two works using the weakest failure detector $L$ in crash-stop asynchronous systems ([63], [33]). In [63] a total order of process’ identifiers and the initial knowledge of the membership is necessary. In [33] set agreement is implemented in systems where the knowledge of $n$ is required.

The failure detector $L$ is defined and implemented for crash-stop message passing systems in [33] and [99]. $L$ is a failure detector defined for crash-stop systems in such a way that it always returns the boolean value false in some process $p_i$, and, if there is only one correct process $p_j$, eventually $p_j$ returns true permanently. Nevertheless, in both implementations the algorithms always output false in all processes in runs where all processes are correct (i.e., in fail-free runs), which are most frequent in practice. This behavior is relevant because the complexity of all algorithms that implement set agreement with $L$ is improved if the number of processes that return true increases.
Chapter 6

Conclusions and Future Work

The agreement problems are well used tools to build fault-tolerant distributed systems. In this thesis, several representative problems have been studied that they are extended from the eponymous distributed systems to the homonymous and anonymous distributed systems. In order to realize this extension, different symmetry breaking methods are proposed.

6.1 Summary of Results

This thesis mainly has six contributions. For fault-tolerant broadcast, the fault-tolerant broadcast algorithms are designed firstly in anonymous distributed systems with reliable communication channels and an implementation of failure detector $\psi$ is also given; in the second part, we provide firstly two non-quiescent but simple algorithms in anonymous distributed systems, and then proposed two classes of anonymous failure detectors $A\Theta$ and $AP^*$, and two quiescent fault-tolerant broadcast algorithms are given finally. For consensus, a new algorithm of consensus with $A\Omega'$ is proposed in anonymous distributed systems with reliable communication channels. For K-set agreement, an algorithm of set agreement and an implementation of failure detector class of $L$ in homonymous distributed system with crash-recovery failure model are given.

1. An impossibility result related to uniform reliable broadcast.

It is impossible to implement uniform reliable broadcast without a majority of correct processes in anonymous distributed systems, no matter with fair lossy or reliable communication channels. This impossibility result stems from the fact that it is impossible for a process to confirm a message has been received by at least one correct process before the uniform reliable broadcast of it. This is due to two reasons: asynchrony and anonymity.
2. Two new anonymous failure detector classes for fault-tolerant broadcast.
   We propose two new classes of anonymous failure detectors $A\Theta$, $AP^*$. $A\Theta$ is aimed to take place of the condition of a majority of correct processes of uniform reliable broadcast; $AP^*$ is designed to make the non-quiescent fault-tolerant broadcast algorithms to be quiescent. Furthermore, all the implementation algorithms of these failure detectors are also given.

3. New implementation algorithms of fault-tolerant broadcast abstraction in anonymous asynchronous distributed systems.
   The fault-tolerant broadcast algorithms are proposed in anonymous distributed systems with reliable communication channels firstly, and then in anonymous distributed systems with fair lossy communication channels.

4. An implementation algorithm of anonymous failure detector class of $A\Omega'$.
   We proved that failure detector $A\Omega'$ is strictly weaker than $A\Omega$. Then, it is implemented in anonymous partially synchronous distributed systems.

5. New implementation algorithm of consensus in anonymous asynchronous distributed systems.
   This algorithm is designed in anonymous asynchronous distributed system that enriched with failure detector $A\Omega'$.

6. New implementation algorithm of set agreement in homonymous asynchronous distributed systems.
   The first set agreement and failure detector $L$ algorithms are given in homonymous asynchronous distributed systems where process can crash/recovery and has incomplete knowledge of membership.

### 6.2 Future Work

According to research results of this thesis, the directions of future works can be summarized as follows:

As future work, we have to study other fault-tolerant broadcast services with different properties of delivery (such as FIFO or Causal order).

Another future line is to search the weakest failure detector that allows to implement each type of fault-tolerant broadcast service in asynchronous anonymous systems.

Finally, the complexity analysis is also a good and meaningful direction for future work. The solutions included in this thesis have been focused to prove the possibility results with
algorithms as simple as possible. Hence, we aim to the researchers to study new algorithms that solve the agreement algorithms regarding the performance or efficiency of the anonymous systems.
References


References


Appendix A

Anonymous Failure Detector Class $A\Theta S$ and Its Application to Fault-tolerant Broadcast

A.1 The Failure Detector Class $A\Theta S$

As mentioned in chapter 3, there are two ways to design anonymous failure detectors: the first way is by assigning a temporal identifier to each process; the second way is by using a shared memory that each process has a shared memory between the fault-tolerant broadcast application layer and the failure detector layer. In this section, we take $A\Theta$ as an example to express our idea of how to use shared memory.

It is necessary to mention that the shared memory utilized in this section is different from the one used in the shared memory distributed system model. The shared memory in this section is a local one that only can be write and read by the fault-tolerant broadcast application and the failure detector of each process; however, in shared memory system model, the shared memory is a global one.

The definition of $A\Theta$ in section 3.4.4 of chapter 4 needs to be redefined if each process has a shared variable $MY\_ACK$ which is accessible to both URB and $A\Theta$ algorithms. If the shared variable is used, the output of $A\Theta$ can be expressed as a series of $(tag, tag2)$, where $tag$ is the label of one message, and $tag2$ is the received ACKs (eventually only include ACKs sent by correct processes) of this message.

A.1.1 The definition of the failure detector $A\Theta S$

As said above, we give the variant definition of $A\Theta$, named as $A\Theta S$, as follows:
• **AΘS-completeness**: There is a time after which all processes who generate those output labels of each message are correct processes.

• **AΘS-accuracy**: If there is a correct process then, then at every time, there exists at least one output label of a message is generated by a correct process.

Then, we give the definition of AΘS more formally:

- **AΘS-completeness**: \( \exists \tau \in \mathbb{N}, \forall i \in \text{Correct}, \forall \tau' \geq \tau, \forall x \in a_{\theta i}^\tau: \text{sender}(x) \subseteq \text{Correct} \).

- **AΘS-accuracy**: \( \text{Correct} \neq \emptyset \implies \forall \tau \in \mathbb{N}, \forall i \in \Pi, \exists m, \exists x \in a_{\theta i}^\tau: m.\text{sender}(x) \in \text{Correct} \).

The failure detector outputs all labels of acknowledgment messages broadcast by correct processes. For each process generates a unique ACK label to a fixed message, these labels are equivalent to identifiers of processes when refers to a fixed message.

When a process receives all ACKs of a message \( m \) from at least one correct process (i.e., all ACKs with the same labels as the output of AΘ), it can safely \( \text{URB}_\text{deliver}(m) \). Because this correct process can guarantee all others correct process will receive this \( m \) eventually.

The failure detector can guarantee that all its output ACKs labels includes at least one label is generated by a correct process. Together with a process regenerates a ACK label of \( m \) only after it has received this message \( m \). It is easy to obtain that at least a correct process has receive the message \( m \), which is safe enough to \( \text{URB}_\text{deliver}(m) \).

### A.1.2 The Algorithm of the Failure Detector AΘS

The failure detector AΘS is implemented in a synchronous system model, in which each process has a bounded execute time and communication time.

**Description of the algorithm:**

Each process has three local sets:

- set \( \text{rec}_i \) label, initialize to be empty, records all received \( \text{tag} \) and \( \text{tag2} \).

- set \( \text{label}_i \), initialized to 0.

- set \( \text{Output}_i \), initialized to 0, is the output set of the failure detector.

Every \( \beta \) time, \( p_i \) reads all pairs of \( (\text{tag}, \text{tag2}) \) (as \( \text{label} \)) from the shared memory set \( \text{MY}_\text{ACK}_i \). Then, \( p_i \) broadcast each \( \text{label} \) \( (\text{tag}_j, \text{tag2}_j) \), and set the \( \text{timer}_i \) to \( \Delta \).

When process \( p_i \) receives a \( \text{label}_j \) from process \( p_j \), it will do as follows:
A.2 Fault-tolerant Broadcast in $\text{AAS}_{F_{n,t}}[\Theta S]$

A.2.1 The Algorithm of RB in $\text{AAS}_{F_{n,t}}[\Theta S]$

Description of the algorithm:

Each process initializes its three sets: $\text{MSG}_i$, $\text{RB\_DELIVERED}_i$, $\text{MY\_ACK}_i$ and activates the Task 1 (Lines 1-3). We take a process $p_i$ as an example to simplify the description. When $p_i$ calls $\text{RB\_broadcast}(m)$, it generates a random number as a tag of this message $m$ and inserts $(m, \text{tag})$ into set $\text{MSG}_i$ (Lines 4-6). Therefore, the first message appears in set $\text{MSG}_i$ that $p_i$ begins to broadcast this $m$ as $(\text{MSG}_i, m, \text{tag})$ to all processes in Task 1 (Lines 25-27).

When $p_i$ receives a message $(\text{MSG}_i, m, \text{tag})$, it first check if this $(m, \text{tag})$ has already existed in its $\text{MSG}_i$ set. If not, it inserts this message to $\text{MSG}_i$. Then, there are two cases as follows:

- The first reception of tag (i.e., tag does not exist in $\text{rec\_label}_i$). $p_i$ allocates this tag in $\text{rec\_label}_i$ and records tag2 in this $\text{rec\_label}_i$ (Lines 10, 11).
- Not the first reception of tag, then $p_i$ only record tag2 in $\text{rec\_label}_i$ (Lines 12-14).

When the timer expires, $p_i$ outputs its set $\text{rec\_label}_i$ and sets $\text{rec}_i$ to empty (Lines 16-18).
**Initialization**

1. sets $MSG_i$, $RB\_DELIVERED_i$, $MY\_ACK_i$ empty
2. activate Task 1

**When $RB\_broadcast(m)$ is executed**

3. $tag \leftarrow \text{random}_i()$
4. insert $(m,tag)$ into $MSG_i$

**When $receive_i(MSG,m,tag)$ is executed**

5. if $(m,tag)$ is not in $MSG_i$ then
6.     insert $(m,tag)$ into $MSG_i$
7. end if
8. if $(m,tag)$ is not in $RB\_DELIVERED_i$ then
9.     insert $(m,tag)$ into $RB\_DELIVERED_i$
10. $tag2 \leftarrow \text{random}_i()$
11. insert $(m,tag,tag2)$ into $MY\_ACK_i$
12. broadcast($ACK,m,tag,tag2$)
13. $RB\_deliver(m)$
14. else
15.     broadcast($ACK,m,tag,tag2$)
16. end if

**When $receive_i(ACK,m,tag,tag2)$ is executed**

17. if $(ACK,m,tag,\_)$ is received for the first time then
18.     allocate set $received\_tag2(m,tag)$
19. end if
20. $received\_tag2(m,tag) \leftarrow received\_tag2(m,tag) \cup tag2$

**Task 1:**

21. repeat forever
22.     for every message $(m,tag)$ in $MSG_i$ do
23.         broadcast($MSG,m,tag$)
24.         if $a\_theta_i(tag) \subseteq received\_tag2(m,tag)$ then
25.             delete $(m,tag)$ from $MSG_i$
26.         end if
27.     end for
28. end repeat

---

Fig. A.2 Quiescent algorithm of $RB$ in $AAS\_{F,n}\_i[A\Theta S]$ (code of $p_i$)

- $(m,tag)$ is received for the first time (i.e., $(m,tag)$ does not exist in the $RB\_DELIVERED_i$ set. Reliable broadcast guarantees that all correct processes deliver the same set of values, so when a process receives a message it deliver it at once and save it in $RB\_DELIVERED$).

$p_i$ generates a new random number $tag2$ and insert $(m,tag,tag2)$ into $MY\_ACK_i$ set. Then, $p_i$ broadcast an acknowledge message ($ACK,m,tag,tag2$) of $m$ that is composed by both $tag2$ and label information (reads all outputted labels from its failure detector $AP_i$). Then, $p_i$ $RB\_deliver m$ for one time (Lines 11-16).
• \((m, \text{tag})\) is not received for the first time (i.e., \((m, \text{tag})\) has already existed in the \(RB_{\text{DELIVERED}}_i\) set).

\(p_i\) rebroadcast the same acknowledge message \((ACK, m, \text{tag}, \text{tag}2)\) to overcome the fair lossy channel (Line 18).

When \(p_i\) receives an acknowledge message \((ACK, m, \text{tag}, \text{tag}2)\) from \(p_j\) (could be itself):

• \(p_i\) receives the very first acknowledgment message of \((m, \text{tag})\), \(p_i\) allocate a set \(\text{received}_\text{tag}2(m, \text{tag})\) and put the received \text{tag}2 in this set.

• \(p_i\) receives not the first ACK message, \(p_i\) put the received \text{tag}2 in the \(\text{received}_\text{tag}2(m, \text{tag})\) set.

In task 1, \(p_i\) broadcast every message \((m, \text{tag})\). If the condition that all \text{tag}2 outputted by the failure detector \(A\Theta S\) is a subset of the received \text{tag}2 of \(p_i\) is satisfied, together with the completeness property of \(A\Theta S\), \(p_i\) can delete message \((m, \text{tag})\) from its \(MSG_i\) set (Lines 25-32).

The run of this algorithm is similar to the previous one except the condition of when to delete a pair of \((m, \text{tag})\) from the set \(MSG_i\), which is the key point to make the algorithm to be quiescent. Because the output of failure detector \(A\Theta S\) exist a one to one mapping relationship between an ACK message\((\text{label})\) and the sender of this ACK message. According to the completeness property of \(A\Theta S\), eventually it output all \text{labels} of correct processes. Then, we get the new condition of delete for a process \(p_i\): When exists a pair of message \((m, \text{tag})\) has been \(RB_{\text{delivered}}\) and \(p_i\) has received all acknowledgment messages from the correct processes (i.e., for a \text{tag}, the received \text{tag}2 is equal to the output of \(A\Theta S_i\)). Then, \(p_i\) can delete this \((m, \text{tag})\) from its \(MSG_i\) set to stop rebroadcast this \(m\).

A.2.2 The Algorithm of URB in \(AAS_{F_{n,t}}[A\Theta S]\)

Description of the algorithm:

Each process initializes its four sets: \(MSG_i, MY\_ACK_i, ALL\_ACK_i, URB_{\text{DELIVERED}}_i\) and activates the Task 1 (Lines 1-3). We take a process \(p_i\) as an example to simplify the description. When \(p_i\) calls \(URB\_broadcast(m)\), it generates a random number as a \text{tag} of this message \(m\) and inserts \((m, \text{tag})\) into set \(MSG_i\) (Lines 4-6). Therefore, the first message appears in set \(MSG_i\) that \(p_i\) begins to broadcast this \(m\) as \((MSG, m, \text{tag})\) to all processes in Task 1 (Lines 26-28).

When \(p_i\) receives a message \((MSG, m, \text{tag})\), it first check if this \((m, \text{tag})\) has already existed in its \(MSG_i\) set. If not, it inserts this message to \(MSG_i\). Then, there are two cases as follows:
Anonymous Failure Detector Class $A\Theta S$ and Its Application to Fault-tolerant Broadcast

1 Initialization
2 sets $MSG_i, MY\_ACK_i, ALL\_ACK_i, URB\_DELIVERED_i$ empty
3 activate Task 1

4 When $URB\_broadcast(m)$ is executed
5 tag ← random_i()
6 insert $(m, tag)$ into $MSG_i$

7 When receive($MSG, m, tag$) is executed
8 if $(m, tag)$ is not in $MSG_i$ then
9 insert $(m, tag)$ into $MSG_i$
10 end if
11 if $(m, tag, tag2)$ is in $MY\_ACK_i$ then
12 broadcast($ACK, m, tag, tag2$)
13 else
14 tag2 ← random_i()
15 insert $(m, tag, tag2)$ into $MY\_ACK_i$
16 broadcast($ACK, m, tag, tag2$)
17 end if

18 When receive($ACK, m, tag, tag2$) is executed
19 if $(m, tag, tag2)$ is not in $ALL\_ACK_i$ then
20 insert $(m, tag, tag2)$ into $ALL\_ACK_i$
21 end if
22 When $\exists (m, tag)$ is not in $URB\_DELIVERED_i \land a\_theta_i(tag) \subseteq ALL\_ACK_i(m, tag, -)$ then
23 insert $(m, tag)$ into $URB\_DELIVERED_i$
24 $URB\_deliver(m)$
25 end if

Task 1:
26 repeat forever
27 for every message $(m, tag)$ in $MSG_i$ do
28 broadcast($MSG, m, tag$)
29 if $\exists (m, tag)$ in $URB\_DELIVERED_i \land a\_theta_i(m, tag) \subseteq ALL\_ACK_i(m, tag, -)$
30 delete $(m, tag)$ from $MSG_i$
31 end if
32 end for
33 end repeat

Fig. A.3 Quiescent algorithm of URB in $AAS_{F_n,t}[A\Theta S]$ (code of $p_i$)

- This is not the first reception of $(m, tag)$ (i.e., $(m, tag, tag2)$ has already existed in the $MY\_ACK_i$ set).

  $p_i$ re_broadcast the acknowledge message $(ACK, m, tag, tag2)$ to overcome the fair lossy channel (Lines 11, 12).

- This is the first reception of $(m, tag)$ (i.e., $(m, tag, tag2)$ does not exist in the $MY\_ACK_i$ set).
\( p_i \) generates a new random number \( \text{tag2} \) and insert \((m, \text{tag}, \text{tag2})\) into \( MY_{\text{ACK}}_i \) set. Then, \( p_i \) broadcast this acknowledgment message \((\text{ACK}, m, \text{tag}, \text{tag2})\) to all (Lines 14-16).

When \( p_i \) receives an acknowledgment message \((\text{ACK}, m, \text{tag}, \text{tag2})\) from \( p_j \) (could be itself). If \((m, \text{tag}, \text{tag2})\) does not exist in \( ALL_{\text{ACK}}_i \), then insert it into this set. Otherwise, \( p_i \) does nothing.

If there exists a pair of \((m, \text{tag})\) has not been \( URB_{\text{delivered}} \) and the output of \( A\Theta S_{a_{\theta i}}(\text{tag}) \) is a subset of the received \( \text{tag2} \) set \( ALL_{\text{ACK}}_i(m, \text{tag}, -) \), \( p_i \) insert \((m, \text{tag})\) into \( URB_{\text{delivered}} \) and \( URB_{\text{deliver}} m \) for one time.

In task 1, \( p_i \) broadcast every message \((m, \text{tag})\). If the condition that all \( \text{tag2} \) outputted by the failure detector \( A\Theta S \) is a subset of the received \( \text{tag2} \) set \( ALL_{\text{ACK}}_i(m, \text{tag}, -) \), \( p_i \) can delete message \((m, \text{tag})\) from its \( MSG_i \) set (Lines 25-32).

Each pair of \((m, \text{tag})\) and its ACK message are unique in the anonymous systems, it can be identified without the identifier of process. Together with the accuracy property of \( A\Theta \) that there exists at least one label that generated by a correct process, and this label is also be used as a tag to ACK message, it guarantees that at least one correct process has received the message. If this condition satisfied, a process can \( URB_{\text{deliver}}() \) the message (Lines 22-24).

As the same as RB algorithm, with the completeness property of \( A\Theta \), a pair of \((m, \text{tag})\) can be deleted when \( p_i \) receives all ACK messages (labels) outputted by the failure detector \( A\theta \) of this pair of message.