Implementing Uniform Reliable Broadcast in Anonymous Distributed Systems with Fair Lossy Channels

Jian Tang*, Mikel Larrea†, Sergio Arévalo‡, and Ernesto Jiménez‡§

*Distributed Systems Laboratory (LSD), Universidad Politécnica de Madrid, 28031 Madrid, Spain
Email: jiantang@gmail.com
†University of the Basque Country UPV/EHU, 20018 San Sebastián, Spain
Email: mikel.larrea@ehu.es
‡Universidad Politécnica de Madrid, 28031 Madrid, Spain
Email: {sergio.arevalo, erenes}@eui.upm.es
§Prometeo Researcher, EPN, Ecuador
Email: prometeo@epn.edu.ec

Abstract—Uniform Reliable Broadcast (URB) is an important abstraction in distributed systems, offering delivery guarantee when spreading messages among processes. Informally, URB guarantees that if a process (correct or not) delivers a message \( m \), then all correct processes deliver \( m \). This abstraction has been extensively investigated in distributed systems where all processes have different identifiers. Furthermore, the majority of papers in the literature usually assume that the communication channels of the system are reliable, which is not always the case in real systems.

In this paper, the URB abstraction is investigated in anonymous asynchronous message passing systems with fair lossy communication channels. Firstly, a simple algorithm is given to solve URB in such system model assuming a majority of correct processes. Then a new failure detector class \( A\Theta \) is proposed. With \( A\Theta \), URB can be implemented with any number of correct processes. Due to the message loss caused by fair lossy communication channels, every correct process in this first algorithm has to broadcast all URB-delivered messages forever, which makes the algorithm to be non-quiescent. In order to get a quiescent URB algorithm in anonymous asynchronous systems, a perfect anonymous failure detector \( AP^* \) is proposed. Finally, a quiescent URB algorithm using \( A\Theta \) and \( AP^* \) is given.


I. INTRODUCTION

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 [1]. Best effort broadcast 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 [2], 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 ([3], [4], [5]). 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 \( m \).

The services mentioned above have been extensively studied in the non-anonymous system model, where each process has a unique identifier, usually assuming that communication channels are reliable (if a process \( p \) sends a message to a process \( q \) and \( q \) is correct, then \( q \) eventually receives \( m \)) or quasi-reliable (if a process \( p \) sends a message to a process \( q \), and the two processes are correct, then \( q \) eventually receives \( m \)) [6]. 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 is sent an arbitrary but finite number of times, there is no guarantee on its reception, it can lose an infinite number of messages [7]). In this regard, several works have addressed the construction of reliable channels...
over unreliable channels in non-anonymous systems ([7], [8]).

As far as we know, the first research on anonymous systems was conducted by Angluin [9], which led to the works of Yamashita and Kameda ([10], [11]). Then, several papers appeared in this field, e.g., ring anonymous networks and shared memory anonymous systems ([12], [13], [14], [15]).

In [16], the reliable broadcast abstraction has been studied in anonymous systems assuming reliable channels.

In classic message passing distributed systems, processes communicate with each other by sending and receiving 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 [17]. However, all these rules have to be changed in anonymous systems. In this paper, each process has a broadcast() communication primitive, with which a process can send a message to all processes (including itself).

Our Contributions This work is devoted to the study of the Uniform Reliable Broadcast (URB) abstraction in anonymous asynchronous message passing distributed systems where processes may crash and communication channels are fair lossy. There are four main contributions in this paper:

- A simple, non-quiescent uniform reliable broadcast algorithm in such a system model assuming a majority of correct processes, which proves that URB can be solved in anonymous asynchronous message passing distributed systems.
- An impossibility result on solving URB without a majority of correct processes.
- Two new classes of anonymous failure detectors AΘ and AP*.
- A quiescent uniform reliable broadcast algorithm using AΘ and AP*, which does not require a majority of correct processes.

Roadmap This paper is organized as follows. The system model and several definitions are presented in Section 2. A simple and non-quiescent algorithm implementing uniform reliable broadcast is proposed in Section 3 under the condition of a majority of correct processes. Then, in Section 4, an impossibility result on solving uniform reliable broadcast without the condition of a majority of correct processes is given. In order to circumvent this impossibility result and make the algorithm quiescent, two classes of failure detectors AΘ and AP* are proposed in Section 5. Then, a quiescent uniform reliable broadcast algorithm with AΘ and AP* is given in the Section 6. Finally, the conclusions are presented in Section 7.

II. SYSTEM MODEL AND DEFINITIONS

In this paper, the anonymous asynchronous distributed system is considered as a system in which processes have no identifiers and communicate with each other via a completely connected network with fair lossy communication channels. Two primitives are used in this system to send and receive messages: broadcast(m) and receive(m). We say that a process pᵢ broadcasts a message m when it invokes broadcastᵢ(m).

Similarly, a process pᵢ receives a message m when it invokes receiveᵢ(m).

Process The anonymous asynchronous distributed system is formed by a set of n anonymous processes, denoted as Π = {pᵢ}ᵢ=1,...,n, such that its size is |Π| = n, i (1 ≤ i ≤ 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 p₁, ⋯, pᵢ 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. Note that this global clock is an auxiliary concept that we only use it for notation, but 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 not execute any more statements or recover. We also assume that at least one correct process exists in the system (i.e., t ≤ n − 1).

Communication Each pair of processes are connected by bidirectional fair lossy communication channels. Processes communicate among them by sending and receiving messages through these channels. We assume that these channels neither duplicate nor create messages, but may lose messages. In anonymous system, when a process receives a message, it cannot determine who is the sender of this message.

Fair Lossy Channel A channel between two processes p and q is called as fair lossy channel if it satisfies the following properties [18]:

- 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.

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).
Failure Detector  A failure detector is a module that provides each process a read-only local variable containing failure information (may be unreliable) of processes. It can be divided into different classes according to the quality of the provided failure information. It was introduced in [19].

Notation The system model is denoted by $AAS_{F_n,t}[0]$ or $AAS_{F_n,t}[D]$. $AAS_F$ is an acronym for anonymous asynchronous message passing distributed systems with fair lossy communication channels; $0$ 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.

III. IMPLEMENTING UNIFORM RELIABLE BROADCAST IN $AAS_{F_n,t}[t < n/2]$

In this section, a simple implementation algorithm of uniform reliable broadcast under the condition of a majority of correct processes is proposed. The system model of this section is denoted by $AAS_{F_n,t}[t < n/2]$.

As far as we know, the implementation of the URB abstraction in the classic (non-anonymous) asynchronous systems with a majority of correct processes is simple. In order to ensure the URB termination property, the construction relies on one condition: a message $m$ can be locally URB-delivered to the upper application layer when this $m$ has been received by at least one non-faulty process. 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 $m$. Obviously, this condition is also needed to be satisfied in the anonymous distributed systems. However, there is no easy way to identify who is the correct process that has received $m$ in the anonymous asynchronous message passing distributed systems due to the fact that all processes have no identifiers. In order to solve this difficulty, the idea of implementing URB in $AAS_{F_n,t}[t < n/2]$ is as follows: 1) to add a unique label (tag) to each message by its sender before it to be broadcast; 2) to add a unique label (tag_ack) to each acknowledgment message (denoted by $ACK$) when a process receives a message.

With the idea mentioned above, the $URB_{deliver}$ condition can be expressed in an equivalent way: each process can deliver a message $m$ if it has received a majority of distinct $ACK$s of $m$. Together with the condition of a majority of correct processes, it is guaranteed that at least one correct process has received $m$.

Description of the algorithm: Algorithm 1 is the implementation algorithm of uniform reliable broadcast abstraction in $AAS_{F_n,t}[t < n/2]$. In this algorithm, two types of messages are transmitted: $MSG$ (a message needs to be $URB_{delivered}$) and $ACK$ (reception acknowledgment of a message). Each process manages a random function $random()$ and four local sets:

- $MSG_i$, initialized to empty, records all messages that it has received.
- $MY\_ACK_i$, initialized to empty, records all acknowledgment messages of each message generated by itself.
- $ALL\_ACK_i$, initialized to empty, records all acknowledgment messages of each message it has received (from any process).

Then, let us consider a process $p_i$ to simplify the description. At the beginning, $p_i$ initializes all its sets into empty and activates Task 1 (lines 1-3).

When $p_i$ calls $URB_{broadcast}(m)$, it assigns a unique random tag to this message $m$, and inserts this pair of $(m, tag)$ into $MSG_i$ (lines 4-6). Then, this $(m, tag)$ is broadcast forever in the Task 1 to propagate it to all processes (lines 28-32).

<table>
<thead>
<tr>
<th>Algorithm 1 Uniform Reliable Broadcast in $AAS_{F_n,t}[t &lt; n/2]$ (code of $p_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization</strong></td>
</tr>
<tr>
<td>1. sets $MSG_i$, $MY_ACK_i$, $ALL_ACK_i$, $URB_{DELIVERED_i}$ empty</td>
</tr>
<tr>
<td>2. activate Task 1</td>
</tr>
<tr>
<td><strong>When $URB_{broadcast}(m)$ is executed</strong></td>
</tr>
<tr>
<td>4. $tag \leftarrow random()$</td>
</tr>
<tr>
<td>5. insert $(m, tag)$ into $MSG_i$</td>
</tr>
<tr>
<td><strong>When receive($MSG, m, tag$) is executed</strong></td>
</tr>
<tr>
<td>7. if $(m, tag)$ is not in $MSG_i$ then</td>
</tr>
<tr>
<td>8. insert $(m, tag)$ into $MSG_i$</td>
</tr>
<tr>
<td>9. $tag_{ack} \leftarrow random()$</td>
</tr>
<tr>
<td>10. insert $(m, tag, tag_{ack})$ into $MY_ACK_i$</td>
</tr>
<tr>
<td>11. broadcast_i($ACK, m, tag, tag_{ack}$)</td>
</tr>
<tr>
<td><strong>When $URB_{deliver}(m)$ is executed</strong></td>
</tr>
<tr>
<td>18. if $(m, tag, tag_{ack})$ is not in $MY_ACK_i$ then</td>
</tr>
<tr>
<td>19. insert $(m, tag, tag_{ack})$ into $ALL_ACK_i$</td>
</tr>
<tr>
<td>20. broadcast_i($URB_{DELIVERED_i}$)</td>
</tr>
<tr>
<td><strong>Task 1:</strong></td>
</tr>
<tr>
<td>23. if $(m, tag)$ is not in $URB_{DELIVERED_i}$ then</td>
</tr>
<tr>
<td>24. insert $(m, tag)$ into $URB_{DELIVERED_i}$</td>
</tr>
<tr>
<td>25. $URB_{deliver_i}(m)$</td>
</tr>
<tr>
<td>26. end if</td>
</tr>
<tr>
<td>27. end if</td>
</tr>
<tr>
<td><strong>repeat forever</strong></td>
</tr>
<tr>
<td>28. for every message $(m, tag)$ in $MSG_i$ do</td>
</tr>
<tr>
<td>29. broadcast_i($MSG, m, tag$)</td>
</tr>
<tr>
<td>30. end for</td>
</tr>
<tr>
<td>31. end repeat</td>
</tr>
</tbody>
</table>

- $URB_{DELIVERED_i}$, initialized to empty, records all $URB_{delivered}$ messages.
- $MY\_ACK_i$, initialized to empty, records all acknowledgment messages of each message generated by itself.
- $ALL\_ACK_i$, initialized to empty, records all acknowledgment messages of each message it has received (from any process).
When \( p_i \) receives a message \((MSG, m, \text{tag})\) (may come from itself or others process), there are three cases:

- If \( p_i \) receives \((MSG, m, \text{tag})\) from itself for the first time (i.e., if this \((m, \text{tag})\) has already existed in \(MSG_i\), but its ACK message \((m, \text{tag}, \text{tag}, \text{ack})\) does not exist in \(MY\_ACK_i\)). This process will go to execute line 14, generates a random \(\text{tag,ack} \) to tag the acknowledgment message of \((m, \text{tag})\). Then, \( p_i \) inserts this acknowledgment message \((m, \text{tag}, \text{tag,ack})\) into its sets \(MY\_ACK_i\), and broadcasts \((\text{ACK}, m, \text{tag,tag,ack})\) to all processes to acknowledge the reception of \((m, \text{tag})\) (lines 15,16). This \(\text{tag,ack} \) is unique for each pair of \((m, \text{tag})\), which means \(\text{tag,ack} \) cannot be changed for the same pair of \((m, \text{tag})\) once it is generated. The local set \(MY\_ACK_i\) is used to maintain this uniqueness, to distinguish the \(\text{tag,ack} \) generated by itself from the received \(\text{tag,ack} \) from others process.

- If \( p_i \) receives \((MSG, m, \text{tag})\) from others process for the first time (i.e., if this \((m, \text{tag})\) does not in \(MSG_i\), neither its ACK message \((m, \text{tag,tag,ack})\) does not exist in \(MY\_ACK_i\)). It inserts this message into \(MSG_i\) (lines 8, 9). Then, like the first case, \( p_i \) generates a random \(\text{tag,ack} \) to tag the acknowledgment message of \((m, \text{tag})\) (line 14). Then, \( p_i \) inserts this acknowledgment message \((m, \text{tag,tag,ack})\) into \(MY\_ACK_i\), and broadcasts \((\text{ACK}, m, \text{tag,tag,ack})\) to all processes to confirm the reception of \((m, \text{tag})\) (lines 15,16).

- If \( p_i \) has received a \((m, \text{tag})\) already (i.e., if this \((m, \text{tag})\) has already existed in \(MSG_i\) and its ACK message \((m, \text{tag,tag,ack})\) also exists in \(MY\_ACK_i\)), it re-broadcasts the identical acknowledgment message \((\text{ACK}, m, \text{tag,tag,ack})\) to all processes in order to confirm the reception of \((m, \text{tag})\) to overcome the message lost caused by the fair lossy communication channels (lines 11,12).

When \( p_i \) receives an acknowledgment message (denoted 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 acknowledgment messages \((m, \text{tag,tag,ack})\) of \((m, \text{tag})\) (more than \(n/2\) different \(\text{tag,ack} \)), and this \(m \) with \(\text{tag} \) has not been \(URB\_delivered \) yet, then \( p_i \) \(URB\_delivered \) \(m \) for one time (lines 22-25).

**Theorem 1** The algorithm 1 guarantees the property of \(URB\).

The correct proof of this theorem is straightforward, and it can be found in [21].

**Remark:** The algorithm 1 can fulfill a fast \(URB\_delivered()\) of a message due to the property of fair lossy communication channels and the asynchrony of the system. For example, a process may receive a majority of acknowledgment messages \((\text{ACK}, m, \text{tag,tag,ack})\) and \(URB\_delivered \ m \) (according to line 22). Hence, this \(URB\_delivered \) is earlier than the reception of \((MSG, m, \text{tag})\). However, this does not violate the property of \(URB\), even if this fast deliver process crashes after \(URB\_delivered \ m \). Because this fast deliver process has received a majority of acknowledgment messages of \(m \) before 

\[URB\_delivered() \ m, \] which means a majority of processes have received this \(m \) (different processes generate distinct ACKs to the same \(m \)). Because together with the condition that there is a majority of correct processes, it is guaranteed that at least one correct process has received \(m \). Then, this correct process will broadcast \(m \) forever guaranteeing that all correct processes will receive \(m \). If the fast deliver process is correct, it will receive \(m \) eventually from others correct process.

It is necessary to generate a unique \(\text{tag} \) to each \(MSG \) and a unique \(\text{tag,ack} \) to each \(\text{ACK} \) in this algorithm. However, it is possible that one random value can be shared by two messages only if one is \(MSG \) type and another one is \(ACK \) type.

### IV. An Impossibility Result

In this section, it is proved that the assumption of a majority of correct processes in the algorithm 1 is a necessary condition to solve \(URB \) in \(AAS\_F_{n,t}()\) if without any other additional assumption.

**Theorem 2** It is impossible to solve \(URB \) in \(AAS\_F_{n,t}()\) without a majority of correct processes.

**Proof:** The proof is by contradiction, let us suppose there exists an algorithm \(A \) that solves \(URB \) in \(AAS\_F_{n,t}([1 \geq n/2])\). Then we divide all processes in the system into two subsets \(S_1 \) and \(S_2 \), such that \(|S_1| = |n/2|\) and \(|S_2| = |n/2|\). Now, we consider two runs: \(R_1 \) and \(R_2 \).

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

- **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 \(URB\_broadcast()\), and they all \(URB\_delivered \ m \). 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 \(URB\_delivered \ m \).

It is easy to see that all processes of \(S_1 \) cannot distinguish run \(R_2 \) from run \(R_1 \) before they \(URB\_delivered \ m \), 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 2.

### V. Two Failure Detector Classes

In this section, two classes of failure detector are proposed. One is used to circumvent the impossibility mentioned above, one is used to make the algorithm 1 to be quiescent.

**A. Failure Detector \(A\Theta \)**

Following the previous impossibility result, one question appears naturally, that is, what extra information is needed if the 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 \( p_i(i \neq j) \) \( \text{URB\_deliver} \) this \( m \). Thanks to the failure detector that was proposed by S. Toueg, this confirmation can be guaranteed by the usage of the (unreliable) failure information provided by it. In this section, we try to circumvent such an impossibility result by using the failure detector.

In non-anonymous systems, failure detector \( \Theta \) is considered as the weakest one to solve URB. It is defined as that it always trust at least one correct process (accuracy) and eventually every correct process do not trust any crashed process (completeness) [18]. The counterpart of this \( \Theta \) in anonymous distributed system is named as \( A\Theta \). Then, we try to define \( A\Theta \) in the anonymous asynchronous distributed systems.

\( A\Theta \) provides the same failure information as \( \Theta \) if each process has a unique identifier. However, it is impossible to give such information in anonymous systems because each process has no identifier. So, the key point to define \( A\Theta \) is how to identify every process without breaking the anonymity of the system. We are inspired by the definition of failure detector class of \( A\Sigma \), which was introduced by F. Bonnet and M. Raynal [20], to define the \( A\Theta \). This \( A\Theta \) provides each process with a read-only local variable \( a_{\text{theta}} \), that contains several pairs of \( (\text{label}, \text{number}) \), in which one \text{label} represents a temporary identifier of one process and \text{number} represents the number of correct processes who have known this \text{label}. For example, process \( p_j \)'s local variable \( a_{\text{theta}} \) = \{\( (\text{label}_1, \text{number}_1) \), ... , \( (\text{label}_n, \text{number}_n) \)\} if there are \( n \) processes in the system. A \text{label} is assigned randomly to each process without breaking the anonymity of the system due to the fact that each process does not know the mapping relationship between a \text{label} and a process (even itself).

The definition of \( A\Theta \) is given as follows:

\( A\Theta \)-completeness: There is a time after which the output variable \( a_{\text{theta}} \) permanently contains pairs of \( (\text{label}, \text{number}) \), associated to all correct processes.

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

Then we give this definition more formally.

Formal definition of \( A\Theta \):

\[ S(\text{label}) = \{i \mid \exists \tau \in \mathbb{N}: (\text{label}, –) \in a_{\text{theta}}^\tau \} \]

is a set of all processes who have known the \text{label}.

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

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

B. Failure Detector \( AP^* \)

An algorithm is quiescent means that eventually no process sends or receives messages. Hence, it is obvious that the algorithm 1 is a non-quiescent algorithm since every correct process has to broadcast all \( URB\_delivered \) messages forever. However, a quiescent algorithm is more valuable and practical in the real systems. In this section, we try to solve this quiescent problem.

The intuitive idea to obtain a quiescent URB algorithm is to terminate the forever broadcast in the algorithm 1. According to the property of uniform reliable broadcast, this forever broadcast can be stopped when a message has been \( URB\_delivered \) by all correct processes (i.e., delete messages that have been \( URB\_delivered \) by all correct processes from the set \( MSG \)). In order to realize this idea, another failure detector \( AP^* \) is needed to enrich the system model to provide the information of who are correct processes in the system.

Anonymous perfect failure detector \( AP^* \) provides each process with a read-only local variable \( a_p^* \) that contains several pairs of \( (\text{label}, \text{number}) \), which is similar to the failure detector \( A\Theta \).

To be more clearly, the definition of \( AP^* \) is given as follows:

- \( AP^* \)-completeness: There is a time after which the output variable \( a_p^* \) permanently contains pairs of \( (\text{label}, \text{number}) \) associated to all correct processes.

- \( AP^* \)-accuracy: If a process crashes, the \text{label} of this process and the corresponding \text{number} will be eventually and permanently deleted from the output variable \( a_p^* \).

Eventually the number of pairs of \( (\text{label}, \text{number}) \) is equal to the number of correct processes.

Let us define \( AP^* \) more formally:

\[ S(\text{label}) = \{i \mid \exists \tau \in \mathbb{N}: (\text{label}, –) \in a_p^\tau \} \]

which is the set of all processes who have known this \text{label} at time \( \tau \) according to \( a_p^\tau \).

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

- \( AP^* \)-accuracy: \( \forall i, j \in \Pi, i \in \text{Correct}, \exists \tau: \forall \tau' \geq \tau: (\text{label}_j, \text{number}_j) \notin a_p^\tau \).

VI. QUIESCENT UNIFORM RELIABLE BROADCAST IN \( AAS\_F_{n,t}[A\Theta, AP^*] \)

In this section, the anonymous asynchronous distributed system model is enriched by both failure detectors \( A\Theta \) and \( AP^* \), denoted by \( AAS\_F_{n,t}[A\Theta, AP^*] \). The algorithm 2 is the quiescent implementation algorithm of the uniform reliable broadcast abstraction in \( AAS\_F_{n,t}[A\Theta, AP^*] \) under the assumption of any number of processes can crash. We first give a detailed description of it as follows:

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 \text{tag} to this
Algorithm 2 Quiescent Uniform Reliable Broadcast in $AAS_{F_{i,t}}[\mathcal{A}_\Theta, \mathcal{A}^*_\mathcal{P}]$ (code of $p_i$)

1. **Initialization**
   - sets $MSG_i$, $URB\_DELETED_i$, $MY\_ACK_i$, $ALL\_ACK_i$, empty
   - activate Task 1

2. **When $URB\_broadcast_i(m)$ is executed**
   - $tag \leftarrow \text{random}_i()$
   - insert $(m, tag)$ into $MSG_i$

3. **When $receive_i(MSG, m, tag)$ is executed**
   - if $(m, tag)$ is not in $MSG_i$ then
     - insert $(m, tag)$ into $MSG_i$
   - end if

4. **If $(m, tag, tag\_ack)$ is in $MY\_ACK_i$**
   - $labels_i \leftarrow \{\text{label} \mid \text{label}, \in \Theta_i\}$
   - broadcast($ACK, m, tag, tag\_ack, labels_i$)
   - else
     - $tag\_ack \leftarrow \text{random}_i()$
     - insert $(m, tag, tag\_ack)$ into $MY\_ACK_i$
     - $labels_i \leftarrow \{\text{label} \mid \text{label}, \in \Theta_i\}$
     - broadcast($ACK, m, tag, tag\_ack, labels_i$)
   - end if

5. **When $receive_i(ACK, m, tag, tag\_ack, labels_j)$ is executed**
   - if $(m, tag, -, -)$ is not in $ALL\_ACK_i$ then
     - allocate array $label\_counter_i[(m, tag), -]$ and $array\_all\_labels_i[(m, tag), -]$
   - end if

6. **If $(m, tag, tag\_ack)$ is not in $ALL\_ACK_i$**
   - insert $(m, tag, tag\_ack)$ into $ALL\_ACK_i$
   - $all\_labels_i[(m, tag), tag\_ack] \leftarrow labels_j$
   - for each label in $labels_j$
     - $label\_counter_i[(m, tag), label] \leftarrow label\_counter_i[(m, tag), label] + 1$
   - end for
   - else
     - for each label in $labels_j$ but not in $all\_labels_i[(m, tag), tag, tag\_ack]$
       - $all\_labels_i[(m, tag), tag\_ack] \leftarrow all\_labels_i[(m, tag), tag\_ack]$ union \{label\}
       - $label\_counter_i[(m, tag), label] \leftarrow label\_counter_i[(m, tag), label] + 1$
       - end for
   - end if

7. **If $(m, tag)\_tag$ 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 $\mathcal{A}_\Theta$. Then, $p_i$ broadcasts $(ACK, m, tag, tag\_ack, labels_i)$ to all processes to acknowledge the reception of $(m, tag)$ (lines 17-20).

8. **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 $URB\_DELETED_i$, 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).

  p_i allocates an array label_counter_i[(m, tag), –] (used to record the number of every label that received together with (m, tag)), and an array all_labels_i[(m, tag), –] (used to records all label in each ACK message of (m, tag) (lines 23-25).

- p_i receives an ACK message coming from a new process (by checking (m, tag, tag_ack) exists in ALL_ACK_i 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 other processes).

  p_i firstly insert (m, tag, tag_ack) into the set ALL_ACK_i, and labels_j into the array all_labels_i[(m, tag), tag_ack]. After that, for each received label in labels_j, p_i 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).

- p_i 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 2, only one of these two cases can happen. In case 1, for each new label, p_i inserts it into all_labels_i[(m, tag), tag_ack] and increases its count number by 1. In case 2, for each disappeared label, p_i deletes it from all_labels_i[(m, tag), tag_ack] and its corresponding label_counter_i. 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 AΘ that satisfies the condition that the counted number of this label label_counter_i[(m, tag), label] is equal to the outputted number, then p_i checks this m has been URB_delivered or not. If not, p_i URB_deliver m for one time.

In task 1, for each pair of (label, number) in the output of AP_i, if the condition that the counted number of each label label_counter_i[(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_labels_i[(m, tag), –] is equal to the outputted label set of AP_i^∗, then p_i checks this m has already been URB_delivered, then, p_i deletes (m, tag) from its MSG_i set (line 57).

**Lemma 1:** 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, 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 (MSG, m, 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(MSG, m, 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, 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 1.

**Lemma 2:** 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 1, 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 AΘ, 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 2 is correct.

**Lemma 3**: 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 \( MY_{\text{ACK}} \) is used to guarantee that each process broadcasts the identical acknowledgment message to the same \((m, \text{tag})\) (line 18). The set \( URB_{\text{DELIVERED}} \) 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 \( URB_{\text{DELIVERED}} \) (line 47) before \( URB_{\text{deliver}} \) it. With those mechanisms, it is certain that no message \( m \) will be delivered more than once. Hence, the proof is completed.

**Theorem 3** Algorithm 2 is a quiescent implementation of the uniform reliable broadcast communication abstraction in \( AAS_{F_{\alpha}}[\lambda \Theta, A\Phi^*] \).

**Proof**: From Lemma 1, 2 and 3, it is easy to see that algorithm 2 is the implementation of the uniform reliable broadcast. Then, it is only necessary to prove the algorithm 2 satisfies the quiescent property.

An algorithm is quiescent means that eventually no process sends or receives messages. In algorithm 2, it is obvious that the broadcast of \( \text{ACK} \) message is invoked by the reception of \( \text{MSG} \) message. Hence, the proof is reduced to show that the broadcast times of \( \text{MSG} \) message is finite. Moreover, a faulty process only broadcast a finite times of \( \text{MSG} \) message. Hence, the rest of this proof focus on that each correct process broadcast a finite times of \( \text{MSG} \) message.

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

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

- If \( q \) is faulty. Then, \( p \) only receives \( \text{ACK} \) from itself together with the accuracy property of \( A\Phi^* \), the \( \text{label} \) and corresponding number of \( q \) will eventually and permanently removed from the output of \( A\Phi^* \). 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.

**VII. CONCLUSIONS**

In this paper, we have studied how to implement the uniform reliable broadcast abstraction in anonymous asynchronous message passing distributed systems with fair lossy communication channels. A non-quiescent algorithm with the assumption of a majority of correct processes is proposed firstly. Then, in order to obtain a quiescent algorithm and to circumvent the impossibility result of implementing URB without the assumption of a majority of correct processes, two classes of failure detectors are given and used. Finally, the quiescent uniform reliable broadcast algorithm is proposed.

**REFERENCES**


