Literature Review

Andrea Panceri 1884749, Francesco Sudoso 1808353

1 Broadcast communication

Broadcast communication in distributed systems is a communication method where a message is sent from a single source to all other nodes in the system, without re-transmitting one by one. Broadcast communication is used in distributed systems to ensure that all nodes receive the same information, allowing them to maintain consistency and coordination. For example, in a distributed database system, a broadcast message may be used to propagate updates to all nodes, ensuring that they all have the same data. Broadcast communication can be implemented in various ways, depending on the specific requirements of the distributed system. For example, it may use multi-cast, where the message is sent to a specific group of nodes, or it may use flooding, where the message is sent to all nodes in the system indiscriminately. Regardless of the specific implementation, broadcast communication is an essential component of distributed systems, allowing them to maintain consistency and coordination among all nodes. It is typically used in combination with other communication methods, such as point-to-point communication, to provide a robust and flexible communication infrastructure for the system. We have three main types of specification: Best Effort Broadcast (BEB), Regular Reliable Broadcast (RB) and Uniform Reliable Broadcast (URB).

1.1 Best Effort Broadcast

A best effort broadcast, weakest specification of a broadcast communication, is a type of data transmission in which the sender sends data without any guarantees about the reliability or success of the transmission. This means that the sender will transmit the data without verifying whether the receiver has received it or not. Best effort broadcasts are often used in situations where the transmission of data is not critical and can be lost without significant consequences. For example, best effort broadcasts may be used for streaming audio or video over the internet, where some data loss is acceptable as long as the overall quality of the transmission is maintained. Now we reports the interface of the algorithm and the properties:
Events:

- **Request:** $< beb, Broadcast|\ m >$: Broadcasts a message m to all processes.
- **Indication:** $< beb, Deliver|\ p, m >$: Delivers a message m broadcast by process p.

Properties:

- **BEB1:** Validity: If a correct process broadcasts a message m, then every correct process eventually delivers m.
- **BEB2:** No duplication: No message is delivered more than once.
- **BEB3:** No creation: If a process delivers a message m with sender s, then m was previously broadcast by process s.

1.2 Regular Reliable Broadcast

In contrast to a best effort broadcast, a regular reliable broadcast is a type of data transmission in which the sender ensures that the data is reliably received by the intended receiver, you can trust that you deliver the message to everybody or to none. Regular reliable broadcasts are often used in situations where it is important to ensure that the data is accurately and completely received, such as in financial transactions or critical communications. Unlike best effort broadcasts, regular reliable broadcasts may require additional overhead and processing to ensure their reliability, but this can be worth it in situations where the integrity of the data is crucial. Now we reports the interface of the algorithm and the properties:
Events:

- **Request:** $< rb, Broadcast|\ m >$: Broadcasts a message m to all processes.
- **Indication:** $< rb, Deliver|\ p, m >$: Delivers a message m broadcast by process p.

Properties:

- **RB1-RB3** same as properties **BEB1-BEB3** in Best Effort Broadcast.
- **RB4:** Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

1.3 Uniform Reliable Broadcast

URB is an important abstraction in distributed systems, more strong than the reliable broadcast, offering delivery guarantee when spreading messages between processes. URB guarantees that if a process (correct or not) delivers a message m, then all correct processes deliver m, it is based on the idea that we do not deliver as soon as we can, but wait that everybody could potentially deliver the message. Now we reports the interface of the algorithm and the properties:
Events:

- **Request:** $< urb, Broadcast|\ m >$: Broadcasts a message m to all processes.
- **Indication:** $< urb, Deliver|\ p, m >$: Delivers a message m broadcast by process p.

Properties:

- **URB1-URB3** same as properties **RB1-RB3** in Regular Reliable Broadcast
- **URB4:** Uniform agreement: If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

2 Dynamic Distributed Systems

Dynamic distributed systems are systems that involve multiple distributed components that are able to change(or evolve) and adapt over time. It is possible that the available links between the processes change over time or/and the actual processes in the system change over time, all these changes over time could be intentional or due to faults. The main argument for using such systems is that they are able to provide greater flexibility and adaptability compared to traditional, static systems. This can be especially useful in situations where the requirements or constraints of the system are subject to change, or in complex environments where there is a need for coordination and collaboration among multiple components. Additionally, dynamic distributed systems can often be more efficient and scalable than their static counterparts, as they can more easily adjust to changing workloads and conditions. Overall, the use of dynamic distributed systems can help to improve the performance, reliability, and overall effectiveness of complex systems.

3 Byzantine Processes

Byzantine fault tolerance is a term used in distributed computing to describe the ability of a system to continue functioning even when some of its components fail or behave in unexpected ways. In a Byzantine system, there may be multiple byzantine processes that may deviate arbitrarily from the instructions that an algorithm assigns to them (creating fake messages, dropping messages, delay the deliveries, altering the content of messages, ecc.) and act as if they were deliberately preventing the algorithm from reaching its goals. This makes it difficult for the system to reach a consensus or make decisions, as not all components may have the same information or be behaving in the same way. To achieve Byzantine fault tolerance, a system must be able to cope with this uncertainty and continue to operate even in the face of failure. We have the problem that we can't implement a failure detector, we can't detect a correct behavior to a faulty one. One of the key challenges in designing Byzantine processes is to balance the need for robustness and reliability with the need for efficiency and performance. Since Byzantine fault tolerance involves complex algorithms and protocols, it can add overhead and decrease the performance of a system. As a result, Byzantine processes are often used in critical systems where the consequences of failure are high, such as in financial transactions or in systems that control critical infrastructure.

4 Byzantine Reliable Broadcast problem in a dynamic distributed system (reconfiguration)

In a dynamic distributed system, the Byzantine reliable broadcast problem refers to the challenge of reliably broadcasting a message to all nodes in the system, even in the presence of faulty or malicious nodes. In a Byzantine system, some nodes may behave incorrectly or try to disrupt the communication, making it difficult for the other nodes to agree on the contents of the broadcast message. The problem of Byzantine reliable broadcast is particularly relevant in dynamic distributed systems, where the system may be reconfigured or changed while the broadcast is taking place. In these systems, the faulty or malicious nodes may try to exploit the reconfiguration to interfere with the broadcast, making it even more difficult to ensure its reliability. To solve the Byzantine reliable broadcast problem in a dynamic distributed system, it is necessary to use advanced algorithms and protocols that can tolerate the presence of faulty nodes and adapt to changing system conditions. In particular we describe two proposed solutions in the following chapters, and after this we continue the work.

4.1 Goal of the papers

In the two papers the authors study the dynamic problem of broadcast in asynchronous systems with Byzantine faults. It is given an asynchronous message-passing distributed system composed of a set $P = \{p_1, p_2, ...\}$ of interconnected nodes. The authors design protocols that provide interfaces to reconfiguration operations with the existence of Byzantine nodes. In the traction they define a dynamic Byzantine broadcast problem, they solve the dynamic

Byzantine problem by designing a dynamic Byzantine reliable broadcast protocol in a completely asynchronous message-passing system.

5 Dynamic Byzantine Broadcast in Asynchronous Message-Passing Systems by Jing Li, Tianming Yu, Ye Wang and Roger Wattenhofer

5.1 Distributed system model

They consider a distributed system composed of a set P = $\{p_1, p_2, ...\}$ of interconnected nodes. Some nodes in the system are Byzantine. Any node pair p_i and p_j can send messages to each other directly, it is given a complete graph as topology. The transmission mechanism is point-to-point. The communication channels between nodes are authenticated. Nodes can recognize who is the sender of the message when they receive a message. Messages cannot be modified by any third parties if messages are delivered via the authenticated channel between nodes. The system in this paper is completely asynchronous. Nodes only take actions when they are activated by events, such as messages arriving. We assume that the total number of nodes are unbounded and possibly infinite. However, not every node remains active in the system throughout the life of the system. At the beginning of execution time, there is a set of available nodes, Init, which can be accessed by users. Other nodes are initially inactive. Active set Init is known to every node. Nodes can be activated when joining the system or be removed from the system during execution. Once they have been removed from the system, they cannot become active again.

5.2 Problem definition

The first fundamental definition made in the article is that of system configuration: *"A configuration w represents sets of active nodes and removed nodes. It holds two sets: The first set is a set of active nodes, w.active. Nodes in w.active are either active at the beginning of the system or are activated during the execution but not removed from the system. The second set is a set of removed nodes w.removed, which are nodes removed from the system. The size of w is the sum of the size of w.active and the size of w.removed"*. Users can change the system configuration by reconfiguration operations. A set of configuration changes cng contains two subsets. Subset cng.join contains nodes which are activated for joining the system and subset cng.remove contains nodes which will be removed from the system. Another fundamental definition is that of **Liveness conditions**, but to understand the meaning, we report the same preliminary definitions of the paper. At any time t in the execution, these definitions are valid:

- P(t) is defined as the set of pending changes at time t such that a reconfig(cng) was invoked but has not completed.
- B(t) is defined as the set of nodes that are Byzantine (including crashed nodes) by time t.
- $\text{CurConfig}_p(t)$ is the latest configuration that node p keeps at time t, important because the configurations known by each node may be different during the execution.
- A configuration AvaiConfig is available at time t if and only if there exists an active node p such that $\text{CurConfig}_p(t) \subseteq w$.

The liveness conditions hold during the whole execution time of the system:
"at any time t, for any available configuration AvaiConfig in the system at time t, given that $|P(t).join \cup AvaiConfig.active| = N$ *and* $f = |B(t)|$, *we must satisfied that* $N > 3(f + |P(t).remove|)$."
Dynamic broadcast with byzantine faults has two communication and one reconfiguration primitive:

- **broadcast$_p$(m):** Users call this primitive in order to broadcast a message m. Initially this operation is disabled on node p because only nodes that are in the current configuration can broadcast messages;
- **deliver$_p$(m):** When a message m is delivered at node p, this will get an indication with a label l about the sender of the message;
- **reconfing$_p$(cng):** Operation that allows an active node p to add or remove nodes from the system.

In addition the broadcast has two events:

- **enable:** allow the node p to execute operations such as broadcast$_p$ and reconfig$_p$;
- **halt$_p$:** forbids the node p to execute operations such as broadcast$_p$ and reconfig$_p$. After the event, p can still deliver messages that were broadcast before it left the system.

It is important to emphasize that Byzantine nodes cannot add or remove arbitrarily nodes to the system.
The dynamic Byzantine consistent broadcast has four properties:

- **Validity:** If a correct node (active node that follows the algorithm protocol) p broadcasts a message m, then every correct node q which joins the system before and does not leave until after the broadcast of m eventually delivers m.
- **No duplication:** If a correct node p broadcasts a message m with label l, then every other correct node delivers at most one message with label l from p.
- **Integrity:** If a correct node q delivers a message m from sender p and p is correct, then m was previously broadcast by p.
- **Consistency:** If a correct node q delivers a message m with label l from sender p, and another correct node s delivers a message m′ with label l from sender p, then m = m′ .

The dynamic Byzantine reliable broadcast has four properties:

- **Validity, No duplication, Integrity:** same properties as the dynamic Byzantine consistent broadcast.
- **Agreement:** If a correct node q delivers a message m with label l from sender p in configuration w, then every other correct node in configuration w delivers a message m with label l from sender p eventually.

5.3 The weak snapshot abstraction

The weak snapshot abstraction comes to our support for implementing basic functions of the broadcast protocol and it is accessible by a static set P of nodes. Every configuration w has a weak snapshot object ws(w), accessible by every node in w.active. We will denote this object with "ws". It has two operation executable by $\forall$ p $\in$ P: scan$_p$() and update$_p$(cng). These operation satisfy the following properties:

- **Integrity:** Let o be a $\text{scan}_p()$ operation that returns C. Then $\forall$ cng $\in$ C, an update(cng) operation is invoked by some node q prior the completion of o;
- **Validity:** Let o be a $\text{scan}_p()$ operation that is invoked after the completion of an update(cng) operation and returns C. Then, $C \neq \emptyset$;
- **Monotonicity of scans:** Let o be a $\text{scan}_p()$ operation that returns C and let o′ be a $\text{scan}_p()$ operation that returns C′. If o′ is invoked after the completion of o $\Rightarrow C \subseteq C'$;
- **Non-empty intersection:** There exists a change set cng such that $\forall$ scan() operation that return $C \neq \emptyset$, it is stated that cng $\in$ C;
- **Termination:** If some majority M of nodes in P don't crush, then every $\text{scan}_p()$ operation and $\text{update}_p(\text{cng})$ invoked by any node p $\in$ M eventually completes.

The weak snapshot object uses an array of (1,N)-atomic registers as Mem[w] of size $|w.active|$. It also implements three functions:

- **Collect_p(w):** return a set of changes. In particular, for each active node in w.active, it executes a read on the register.
  If the read return a value different from $\emptyset \Rightarrow C = C \cup \text{cng}$. Basically, it returns the set of changes that each active node in the configuration wants to make.
- **Update_p(w):** returns True or False depending on whether the operation was successful. In particular, it first does a $\text{collect}_p(w)$ to read all updates written to the configuration register Mem[w]. If there are no updates, then it writes the value it has as parameter (cng) to Mem[w][p]. This new update will be contained in the next $\text{collect}_p(w)$ $\forall$ p $\in$ w.active.
- **Scan_p(w):** returns a set of changes C. In particular, it first does a collect(w) to read all previous updates for w. If there are no updates in Mem[w] then it returns $\emptyset$, otherwise it does a collect(w) again and returns all previous values.

Basically, when a node p wants to change the configuration w, it does an $\text{update}_p(\text{w,cng})$ in order to propose a set of changes cng. It will learn the configuration changes proposed by the other nodes through a $\text{scan}_p(w)$, which returns a set of changes.

5.4 Dynamic Byzantine Consistent Broadcast

The algorithm has mainly two phases: the broadcast-phase and the check-phase. The check-phase is used to find new configuration updates, while the broadcast-phase is used to broadcast messages, after checking that there are no updates in pending.
The goal of these phases is to ensure that no message is delivered in an old configuration while a configuration update is being performed. Now we show the local **State of nodes**, the local variables of each node.
Local variables for handling the configurations of node p:
ConfigList_p = initially $\emptyset$, contains all configuration which have been visited.
CurrConfig_p = initially Init, contains the latest configuration.

Local variables for handling the messages:
MsgNum_p = initially 0, contains the number of messages broadcasted.
ReceivedM_p = initially $\emptyset$, contains the ECHO messages received.
ValidateM_p = initially $\emptyset$, contains the messages validated by a quorum, waiting for the delivery.

DeliveredM$_p$ = initially $\emptyset$, contains the messages delivered.

It is possible to divide the algorithm the Dynamic Byzantine Consistent Broadcast into 4 parts:

- **Broadcasting messages:** The operation broadcast$_p$(m) firstly invokes the function Traverse$_p$($\emptyset$, m) in order to find the latest configuration in the system and then broadcast the message m. We have $\emptyset$ as argument of Traverse$_p$ because this execution does not change the configuration but only scans and then broadcast the message. After broadcasting the message, the variable MsgNum$_p$ will be increased by 1.

- **Reconfigurations:** The operation reconfig$_p$(cng) firstly invokes the function Traverse$_p$(cng,$\perp$) in order to find the latest configuration in the system and applies cng to the system. We have $\perp$ as an argument of Traverse$_p$ because this execution does not broadcast any messages. After applying the changes found in cng, each node will send to every other node in the configuration a message labeled with "Join", in order to communicate the imminent inclusion of new nodes in cng.

- **Delivering messages:** A passive procedure that is triggered when nodes receive broadcast messages from other nodes. After a message m with tag SEND is received in configuration w by node s, node p will broadcast a message m with ECHO tag to all other nodes in configuration w. When node p reaches the quorum of $\frac{2*|w.active|}{3}$ ECHO messages with the same message m, same sender s, in the same configuration w and same message number MsgNum, validates the message and stores it in the set ValidatedM$_p$. After this node p starts to find the latest configuration in the system by invoking Traverse$_p$($\emptyset$, $\perp$) . Message m received by p in configuration w is not delivered immediately because w could be a fake configuration and message m may not be broadcast in the real system. Node p delivers message m if it has visited the configuration w where m was broadcast. Node p checks all validated messages by calling function ReceiveMsg$_p$() when it visits a new configuration or it receives an ECHO message.

- **Traversing the graph of configs:** The core and most complex function of the algorithm. Thanks to the weak snapshot, it is possible to organize all configurations in a directed graph in which the configurations are the vertices, and edges represent the transition from one configuration to another, i.e. when there is an update$_p$(w,cng) from one configuration to another. The node that calls this function will get the latest configuration available in the system. For this reason, Traverse$_p$() is invoked by all operations(broadcast$_p$(), reconfig$_p$()) and when p receives messages from others, in this way they can keep up to date with the current state of the system.
  The function starts the scan from the configuration found in the local variable CurConfig and while it travers all the directed graph, it collects all the changes inside the variable desiredConfig. If the collected set of changes (which are in desiredConfig) are not included in the graph, the algorithm will include them invoking the update$_p$(w, desiredConfig $\setminus$ w) function and then the weak snapshot object of desiredConfig will be connected to the graph.
  To traverse the directed graph, the algorithm uses the Dijkstra algorithm with a modification such that it is able to modify the graph during the execution. Specifically, the function starts by initializing a configuration set Front = CurConfigp and at each iteration Traverse$_p$ visits the closest configuration in Front, i.e. the one with the smallest size in the set, and removes that vertex from Front after visiting. When the function

visits a configuration w, it first checks that node p is active (is present in w.active) and then delivers the messages that p has in ValidatedM$_p$ by invoking the ReceiveMsg$_p$() function. But if w is not up-to-date on the changes found during traversal, then node p will invoke the update$_p$() function to apply the update to w's weak snapshot object and construct an edge from w to desiredConfig. After that, the algorithm performs a scan$_p$(w) because if there has been an update to weak snapshot object of w, the scan function will return it and assign it to ChangeSets. Depending on the return value of the scan, we can distinguish two cases:

– if there has been an update $\rightarrow$ ChangeSets $\neq \emptyset$.
  So, for each change in ChangeSets the algorithm will add the change c to the desiredConfig and to the Front(for the furthering visit).
– if there hasn't been an update $\rightarrow$ ChangeSets = $\emptyset$.
  So, p invokes BroadcastInConfig(w,m) which broadcasts the message m and scans the weak snapshot object of w again.

Finally, the Traverse$_p$() function terminates the execution by returning desiredConfig.

A problem in the algorithm is when multiple nodes invoke update(w) at the same time, concurrent scan(w) might see different outgoing edges from w, which implies that different nodes might keep different configurations. This problem is covered by the non-empty intersection property of weak snapshot objects; it ensures that nodes will never work on different branches of the digraph. Actually, at least one outgoing edge is returned by all scan(w) functions and the destination of this edge will be visited by all traverse() functions. This fact enables us to define a totally ordered subset of **established configurations**. Important definition to this problem is **Sequence of Established Configurations:**
The unique sequence of established configurations ε is constructed as follows:

- *The first configuration in ε is the initial configuration Init;*
- *if w is in ε, then the next configuration after w in ε is w' = w+cng, where cng is an element chosen arbitrarily from the intersection of all sets C $\neq \emptyset$ returned by some scan(w) operations in the execution.*

Also valid these considerations: "*Whenever BroadcastInConfig$_p$(w, $\star$) is invoked, w is an established configuration*" and "*Let T be an execution of Traverse$_p$() and initConfig be the value of CurConfigp when node p starts this execution. Let desiredConfig be the value of CurConfig$_p$ when this execution terminates. Then Traverse$_p$() visits all configurations from the sequence of established configurations between initConfig and desiredConfig*". This algorithm with the assumption that has been descriptive before satisfies the properties of the byzantine consistent broadcast with the problem of reconfiguration.

5.5 Dynamic Byzantine Reliable Broadcast

It is possible to build a reliable broadcast based on the precedent protocol. We must change the consistent broadcast protocol that uses the authenticated echo broadcast algorithm to the authenticated double-echo broadcast algorithm that has two ECHO steps. In the first ECHO step, node p sends message m with ECHO tag to other nodes when it accepts the message m with SEND tag from node s. In the second step, if node p receives message m with ECHO tag, it stores it in ReceivedM$_p$. If reach the quorum of $\frac{2*|w.active|}{3}$ copies of the message m with ECHO tag from other nodes in configuration w are stored in ReceivedM$_p$, node p sends message m with READY tag to other nodes in configuration w. Nodes broadcast messages with a READY tag to respond to the first echo. If node p receives a message with a READY tag, it stores it in ReadyM$_p$. If reach a quorum of $\frac{|w.active|}{3}$ copies of the message m with READY tag from other nodes in configuration w are stored in ReceivedM$_p$, node p sends message m with READY tag to other nodes in configuration w. If reach a quorum of $\frac{2*|w.active|}{3}$ copies of the message m with READY tag from other nodes in configuration w are stored in ReceivedM$_p$, node p validates message m in configuration w and stores it in ValidatedM$_p$, for further delivery. This modification satisfied the precedent properties of the consistent broadcast and also the agreement property, and finally achieved a correct protocol for a Dynamic Byzantine Reliable Broadcast in an asynchronous message passing system.

6 Dynamic Byzantine Reliable Broadcast by Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet,Dragos-Adrian Seredinschi and Andrei Tonkikh

6.1 Distributed system model

The model proposed is composed of a universe U of processes, subject to Byzantine failures. Processes that are not subject to failures are correct. We assume an asymmetric cryptographic system. Correct processes communicate with signed messages: prior to sending a message m to a process q, a process p signs m, labeled $< m >_{\sigma_p}$, this implies that the model use authenticated links. Upon receiving the message, q can verify its authenticity and use it to prove its origin to others (non-repudiation). The system U is asynchronous. It is assumed that communication is reliable, i.e., every message sent by a correct process to a correct process is eventually received, implies that it is used perfect point to point authenticad links. It is assumed a global notion of time, outside the control of the processes. The processes can join or leave the system dynamically.

6.2 Problem definition

Dynamic Byzantine Reliable Broadcast (DBRB) interface has three operations and one callback:

- **dbrb-join:** used by a process outside the system to join.
- **dbrb-leave:** used by a process inside the system to leave.
- **dbrb-broadcast(m):** used by a process inside the system to broadcast a message m.
- **dbrb-deliver(m):** this callback is triggered to handle the delivery of a message m.

A process can be in the system initially, Init, or can join the system with a *dbrb-join*. A process is considered a participant of the system if it has not yet invoked *dbrb-leave*. Particularly important is that in the interval time between the invocation of *dbrb-leave* and the retunt the process it is no longer a participant of the system. Join and leave operations can be invoked once. A correct process must follow these rules:

- A *dbrb-join* operation can only be invoked if the process is not participating in the broadcast system.
- Only a participating process can invoke a *dbrb-broadcast(m)* operation.
- A *dbrb-deliver(m)* callback can be triggered only if a process has previously joined but has not yet left the system.

- A *dbrb-leave* operation can only be invoked by a participating process.

Two assumptions must be introduced to ensure the correctness of the algorithm:

- **Assumption 1:** In every execution, the number of processes that want to join or leave the system is finite. (Finite number of reconfiguration requests);
- **Assumption 2:** Initially, at time 0, the set of participants is nonempty and known to every process in U.

The aim of first assumption is that a new reconfiguration requests will be made in a finite time, ensuring that it will be completed at some point. The assumption 2, instead, is defining the state at the beginning of the system and guarantees that all processes have the same starting conditions. In addition, we make two other assumptions that are more common in broadcast implementations:

- The adversaries that are in the system are **not able** to subvert cryptographic primitives;
- A weak broadcast primitive is available, so if a correct process broadcasts a message m, then every correct process eventually delivers m. It is done by a kind of **gossip protocol**.

The dynamic Byzantine reliable broadcast has six properties:

- **Validity:** If a correct participant s broadcasts a message m at time t, then every correct process, if it is a participant at time $t' >= t$ and never leaves the system, eventually delivers m.
- **Totality:** If a correct process p delivers a message m at time t, then every correct process, if it is a participant at time $t' >= t$, eventually delivers m.
- **No duplication:** A message is delivered by a correct process at most once.
- **Integrity:** If some correct process delivers a message m with sender s and s is correct, then s previously broadcast m.
- **Consistency:** If some correct process delivers a message $< m >_{\sigma_\text{p}}$ and another correct process delivers a message $< m' >_{\sigma_\text{p}}$ then m = m′.
- **Liveness:** Every operation invoked by a correct process eventually completes.

In addition DBRB has this non-triviality property: *No correct process sends any message before invoking dbrb-join or after returning from dbrb-leave operation.*

6.3 Building Blocks

In order to simplify the description of the algorithm, the author describes the main components that the algorithm uses. In particular, there are two basic elements:

- **Change:** We define a set of system updates change = { +, - } × U, where the tuple $< +, p >$ or $< -, p >$ indicates that process p asked to join or leave the system. Due to the asynchrony, two processes may concurrently consider different sets of system participants to be valid. To avoid these conflicts, we will use the view abstraction, which defines the system membership through the point of view of the process at a specific point in time.

- **View:** A view v consists of two variables:
  - *v.changes:* which is a set of changes;
  - *v.members:* which is the set of the view membership. It is defined as v.members = $\{$p $\in$ U : $< +, p > \in$ v.changes $\wedge < -, p > \notin$ v.changes$\}$

  The purpose of views is to provide each correct process with an object that it can use to enter all changes that have been correctly executed and that p has observed. The protocol ensures that all valid views are comparable. Formally, v1 $\subset$ v2 means that v1.changes $\subset$ v2.changes, we say that v2 is more recent than v1. Two different views are comparable if one is more recent than the other, otherwise they conflict. We assume that the initial view is known. A valid view v must be equipped with a quorum system of size v.q = $|v|- \lfloor \frac{|v|-1}{3} \rfloor$. In particular, it must be assumed that in each correct view v, the number of Byzantine processes is f $\leq \lfloor \frac{|v|-1}{3} \rfloor$ and at least one quorum in v consists of correct processes. Consequently, the number of processes in view v must satisfy the following inequality: $v \geq 3f + 1$. The quorum of messages is equal to $|ACK| > \frac{2*|v|}{3}$.

- **Sequence of views:** A sequence seq is a set of comparable views.

- **Reliable multicast:** The algorithm is built and based on reliable Byzantine broadcast protocol. Indeed, the functions "R-multicast" and "R-delivery" make use of this protocol for broadcasting and delivering a message.

6.4 Dynamic Byzantine Reliable Broadcast

We start with the local **State of nodes**, the local variables of each node.
Local variables of process p:
cv = v$_0$, contains current view, v$_0$ is the initial one.
RECV = $\emptyset$, contains the set of pending updates, join or leave.
SEQ^v = $\emptyset$, contains the set of proposed set to update v.
LCSEQ^v = $\emptyset$, contains the last converged sequence for update v.
FORMAT^v = $\emptyset$, replacement sequence for view v.
cer = $\perp$, contains the message certificate for m.
v$_{cer}$ = $\perp$, contains view in which the certificate is collected.
allowed_ack = $\perp$, indicate the set of messages allowed to be acknowledged.
stored = false
stored_value = $\perp$
can_leave = false, indicate that a process is allowed to leave.
delivered = false, indicate if m is delivered or not.

Variable for every process q $\in$ U and every valid view v:
acks[q, v] = $\perp$.
Σ**[q, v] =** $\perp$.
deliver[q, v] = $\perp$.
State = $\perp$, contain the state of the process.

We now describe the two fundamental operations that have introduced in the precedent section, which are used to manage the system's dynamicity (i.e join and leave) and the sub-protocol that is crucial for the reconfiguration problem:

- **DBRB-JOIN:** After a correct process invokes the dbrb-join, the first thing it does is to update its local variable cv, so that it obtains the most recent view v. The operation of joining the system follow four steps:
  - p starts broadcasting the $< RECONFIG, < +, p >>$ message to each member of cv, so that it can alert all others of the join request. These two steps, i.e. updating the view to the most recent and broadcasting the reconfiguration message, occur cyclically until the **joinComplete event** is triggered or when a quorum is reached for some view v in which the message was previously broadcast.
  - once each member of cv receives the reconfiguration message, they each propose a new view that is an extension of the previous one and includes the new process. The new proposal will then be inserted into the sequence of views SEQ^v and subsequently broadcast to all cv members via a PROPOSE message.
  - once a process receives the PROPOSE message with the new proposal, it first checks if it is a valid propose and then updates its own proposal. At this point two things can happen:
    - there are conflicts between the proposal of the process and the received one. In this case, the process creates a new proposal that contains its last converged sequence for the view and a new view formed by the union of the most recent views of two proposals.
    - there aren't conflicts between the proposal. In this case, a new proposal formed by the union of the new proposal and the process one.
  - any other correct member in cv waits until **PROPOSE** messages reaches a quorum of processes. After that, the corrects members will update their last converged sequence and start to disseminate a CONVERGED message.
  - each correct process q in cv waits to gather matching **CONVERGED** messages from a quorum of processes. Then after reaching the quorum, q broadcasts an $< INSTALL, v' >$ message specifying the view that should be replaced(for example v) and the last recent view of the new sequence of views(for example w) and the entire new sequence seq'. This message is broadcast to all process that are members of views v or w. Upon an **INSTALL** message is received by any correct process q, it updates its current view to w. At this point, we must distinguish four cases:
    - **q was a member of v:** Firstly, q checks if its current view cv $\subset$ w. In this case, the process will not process any messages that are in his *State* variable, such as PREPARE,COMMIT and RECONFIG messages. This happen because the process have an old view, so it first needs to update its state in order to be align with the members of the new view w.
    - **The view w is more recent that the current view cv of q:** The process q wait an amount of STATE-UPDATE messages equal to the quorum and after it starts to elaborate them. This kind of messages have different type of information inside:
      - the permission to acknowledge a message contained by the variable *allowed_ack*;
      - the stored value of a message contained by the variable *stored_value*;
      - the reconfiguration requests observed by a process.

      All these information are essential because when a process receiving STATE-UPDATE messages it may not know whether the new view is created by the correct protocol or by faulty process.

    - **q is a member of w $\supset$ cv:** In this case q updates and installs the current view if the INSTALL message dones't containt any other views that are more recent than cv.
    - **q is not a member of w $\supset$ cv:** In this case q is leaving the system. Then, for ensure the totality propriety of the DBRB, it executes the View Discovery protocol and disseminate a COMMIT message to all process in cv.

- **DBRB-LEAVE:** This operation executed by a correct process p follow almost the same steps of dbrb-join, except for the fact that p still executes its "duties" in dbrb until dbrb-leave returns.

- **The view Discovery Protocol:** We have that the various views created during an execution of the algorithm form a sequence. We have a subprotocol denominate **View Discovery** that give information on the sequence of view created so far in the execution. We have that any sequence of views start from the the inizial one v_0, and each view of the sequence is connected to the next by an install message. A process participant in the system constantly execute the View Discovery subprotocol, and learns which sequences of views are trusted by other processes. This subprotocol enables processes joining and leaving the system to learn about the current membership of the system, and it is connected to the consistency, validity and totality properties since it give fundamental information about views "instantiated" by the protocol and associated quorum systems.

Now the two main operation for send the message to all participants:

- **DBRB-BROADCAST(m):** A correct process p that invokes *DBRB-BROADCAST(m)* first disseminates a prepare message $m_{\text{prepare}} =< PREPARE, m, cv_{\text{p}} >$ to every participant p' in the current view cv_p, only if current view cv_p is installed by p, otherwise p wait to install some view then disseminate the PREPARE message. When a correct process q receives this PREPARE message, q checks if cv_p is equal to the current view of q cv_q, if it is, q checks whether it is allowed to send an *ACK* message for m, and if it is sends an $m_{\text{ack}} =< ACK, m, \sigma, cv_{\text{q}} >$ message to p. Once p collects a quorum of matching *ACK* messages for m with the same view v, p constructs a message certificate cer=Σ out of the collected signatures σ, then creates $m_{\text{commit}} =< COMMIT, m, cer, v_{\text{cer}} >, cv_{\text{p}} >$ and disseminates m_{commit} to every participant of cv_p. Like with m_{prepare}, p disseminate the COMMIT message to members of cv_p if cv_p is installed by p, otherwise wait to install some view and then disseminates m_{commit}. When any correct process q receives a m_{commit} message, checks that cv_p is equal to the current view of q and that the certificate is valid, then q stores m and sends $m_{\text{deliver}} =< DELIVER, m, cv_{\text{q}} >$ to p as an approval that p can deliver m, but also disseminate m_{deliver} to all members of view cv_q in order to deliver m itself.

- **DBRB-DELIVER(m):** Once any process q collects a quorum of matching deliver messages with the same view v, q triggers *DBRB-DELIVER(m)*. It is important that the PREPARE and ACK phase can be successfully executed in some view v, whereas the COMMIT and DELIVERY phase can be executed in some view v' $\supseteq$ v. This is the reason why v_{cer} is included in a commit message, representing the view in which a message certificate is collected.

7 Comparison and comments

7.1 Introduction

So before starting the comparison on the actual content of the two papers, we want to analyze the form which is very different. The first paper, written by the same author of the course book, addresses the reader not assuming that he knows the subject, therefore tries to be as exhaustive as possible and not use too many mathematical formulas, starting with an accurate description of the problem and explaining all definitions and assumptions. It also does not immediately report the algorithm for Dynamic Byzantine Reliable Broadcast but first shows the implementation of the consistent, so as to arrive at the more complex problem using an intermediate step, so that the reader understands the algorithm more easily. In the second paper the level of knowledge required is higher, many assumptions and definitions are taken for granted, even when they are important at the end of the implementation, the description of the problem is very concise, furthermore the Reliable algorithm is immediately introduced, making it very complex for the reader to deduce how it works. Other considerations can be made on the very strong use of mathematical formalisms on the second paper, which certainly make the text more correct from a theoretical point of view, but perhaps make the text too heavy. In terms of structure, the two papers are fairly in line, they are structured in a fairly similar way.

7.2 Distributed System Model

Both papers, before describing the implementation of the algorithm, offer a description of the system model that is used and that serves as the basis of the algorithm. The main difference between the two models is that in the first solution, it is defined that the total number of nodes is unbounded and possibly infinite, whereas in the second solution, it is not explicitly specified, but can be assumed.
Another point to note is how in both the first and second papers, a node that performs the leave operation can no longer return to the system with the same identity via a join: this is not explicitly specified in the second paper.
Finally, the system is asynchronous for both algorithms and makes use of authenticated P2P links.

7.3 Problem definition and assumptions

All two papers define several basic assumptions, which allow the correct definition and execution of the proposed algorithm. In particular, the two main assumptions are that:

- for each execution, there is a finite number of reconfigurations;
- at the initial time, there is a set of active processes that is known to all nodes in the system.

As regards the properties guaranteed by the two implementations, both define the liveness property, but in a different way: in the first, it consists in having at each instant of time t of the execution, a number of active processes equal to $N \geq 3f + 1$, while in the second, the liveness property consists in the fact that each reconfiguration eventually terminates.
The liveness condition of the first paper is also used in the second paper, with a more mathematical notation, but with the same value, i.e. the number of processes in the system should satisfy $N \geq 3f + 1$
Finally, both DBRB implementations satisfy the same properties: validity, no duplication,

integrity and consistency. In addition, the propriety of the "agreement" of the first paper is the "totality" propriety of the second paper, instead the "liveness" propriety of the second DBRB is also guaranteed in the first by the "termination" propriety of the weak snapshot object.

7.4 Components used by the algorithm

The first paper uses a configuration system, cng, with an accurate definition of the active and removed nodes, which we have already talked about, while the second uses a system of views, which in the end are very similar, but there are some fundamental differences in their management. In the first, the configurations are represented through a directed graph, therefore algorithms related to graphs such as Dijkstra are used, while the second does not explicitly use an abstraction, but a list of views is used, because the text states that the installed views form a sequence. The configuration problem, central to the two algorithms, is handled with two different approaches, in the first paper a weak snapshot object is used which makes use of (1,N)-atomic registers, while in the second there is no use of this separate protocol, but only a weak broadcast primitive is used which acts as the static counterpart of the broadcast seen in class, and therefore a system such as gossip is used to manage the diffusion of configuration updates.

7.5 Algorithm implementation

Regarding the implementation of the algorithm there are many similarities in the two papers. Both define a Dynamic Byzantine Reliable Broadcast(DBRB) which uses an authenticated double-echo broadcast algorithm, also seen in class, to guarantee the robustness and correctness of the algorithm in the presence of Byzantine processes. From a cryptographic point of view, the first paper omits the implementations that serve to guarantee authentication and therefore the integrity of the messages, while in the second, all the signatures and all the checks that must be carried out are specified in each phase of the algorithm to ensure that the messages that will be delivered have not been manipulated by some adversary. From the point of view of a basic reader, the omission of these implementations allow an easier and more understandable reading, while in the second, with the abuse of these notations they make it difficult to understand the text, but on the other hand it certainly turns out to be more correct.
The first parer to update the configuration uses the Traverse, while the second quite specularly uses the view discovery protocol to update the current view. Furthermore, the operations for inserting and removing a node, Join and Leave, are almost identical in both DBRB. Even the broadcast and delivery operations are similar in the two proposed DBRB algorithms. In particular, the main difference that we have noticed between the broadcasts of the two implementations is that in the first the current state of the system is checked by invoking the Traverse function and in the same function the message is sent to all the currently active processes, while in the second algorithm the broadcast first check whether the current view has been installed by all processes and after disseminate the message to all the member of the current view, without however using an auxiliary function. This shows how central the Traverse is to the first paper. Instead for the delivery, as mentioned, is used a double echo, that consist in two phases where it is ensured that the broadcast satisfies the agreement property. For satisfy the agreement and the integrity it must be used a quorum in the two phases. The quorum used by the two algorithms is the same, to complete a phase the first algorithm wait to receive at least $\frac{2*|w.active|}{3}$ and the second algorithm at least $\frac{2*|v|}{3}$ reply messages, only difference is that in the first algorithm to speed up the reach of the quorum

in the second echo, each process do another broadcast of the second echo when $\frac{|w.active|}{3}$ of reply messages is reached.

8 Addition

In the first paper a weak snapshot abstraction is used, which as mentioned in the paper is a weakened version of an already existing abstraction, in particular the one defined in the DynaStore. DynaStore is a distributed storage system that uses the weak snapshot abstraction to provide consistent, fault-tolerant access to data in a distributed system. The weak snapshot abstraction is a technique for capturing the state of a distributed system at a particular point in time, allowing nodes in the system to read and write to the data while still maintaining consistency. In DynaStore, the weak snapshot abstraction is implemented using a combination of version vectors and vector clocks, which are used to track the history of updates to the data and to detect and resolve conflicts between updates made by different nodes. This allows DynaStore to provide consistent, fault-tolerant access to data in the face of Byzantine faults, such as nodes behaving maliciously or failing to communicate correctly with other nodes in the system. One of the key benefits of DynaStore is that it allows nodes in the system to read and write to the data concurrently, without the need for locking or other synchronization mechanisms. This makes it well-suited for use in distributed systems that require high performance and scalability, such as cloud computing environments.
In addition a precedent solution was proposed by Guettaoui et al. that publish a dynamic solution for DBRB in a paper called "Dynamic Byzantine Reliable Broadcast". In his approach, the nodes in the system are organized into a dynamic overlay network, and message dissemination is facilitated using a combination of flooding and random sampling techniques. The protocol is designed to adapt to changes in the system, such as the arrival or departure of nodes, and can tolerate up to one third of the nodes being Byzantine without affecting the reliability of the broadcast.
The solutions proposed by the papers that we have compared have very important real-life applications such as maintaining caches in cloud services or in a publish-subscribe network. In this application dynamic byzantine reliable broadcast can be used to efficiently disseminate updates or changes to the cache or to the published data to all of the relevant nodes in the system. Another important application for DBRB protocols is that of implementing decentralized online payments, also known as cryptocurrencies. In a cryptocurrency system, each node in the network maintains a copy of the ledger, which records all of the transactions that have taken place. To ensure the integrity of the ledger, it is important that all of the nodes receive and process the transactions in the same order. DBRB can be used to broadcast transactions to all of the nodes in the network in a reliable and fault-tolerant manner, enabling the nodes to reach consensus on the state of the ledger. There are both dynamic and static solutions for implementing DBRB. Dynamic solutions, such as the two that we have compaired, are designed to adapt to changing conditions in the system and can be more flexible in terms of fault tolerance and performance. Static solutions, on the other hand, are based on predetermined quorums or sets of nodes that are responsible for facilitating the broadcast, and may have lower overhead and faster performance compared to dynamic solutions.

9 Final comment and open issue

One open issue with DBRB is the need for a centralized trusted third party, also known as a ”bulletin board” to facilitate the broadcast. This centralization can be a vulnerability and a point of failure in the system. There has been research on developing DBRB protocols that do not require a centralized bulletin board, but these approaches can be more complex and may have lower performance compared to protocols that use a bulletin board.
Another open issue with DBRB is the trade-off between fault tolerance and performance. In general, increasing the fault tolerance of a DBRB protocol will come at the cost of reduced performance, and vice versa. Finding the right balance between fault tolerance and performance is an ongoing challenge in the design and implementation of DBRB protocols. In a Dynamic Byzantine Reliable Broadcast (DBRB) protocol, a bulletin board is a centralized trusted third party that is responsible for facilitating the broadcast of messages among the nodes in a distributed system. The bulletin board is typically implemented as a separate server that is responsible for receiving and storing messages from the nodes, and for providing a list of the messages to each node upon request.
The role of the bulletin board is to ensure that all non-faulty nodes receive the same messages in the same order, even if some of the nodes are faulty or behaving maliciously. To achieve this, the bulletin board uses a variety of techniques, such as cryptographic signatures and message authentication codes, to verify the authenticity and integrity of the messages it receives. One potential issue with using a bulletin board in a DBRB protocol is that it can be a point of failure or a single point of attack in the system. If the bulletin board goes down or is compromised, it can disrupt the reliable broadcast of messages. There has been research on developing DBRB protocols that do not rely on a centralized bulletin board, but these approaches can be more complex and may have lower performance compared to protocols that use a bulletin board.
There have been several papers that have discussed the use of a bulletin board in Dynamic Byzantine Reliable Broadcast (DBRB) protocols. Here is an example:
”A Scalable Protocol for Dynamic Byzantine Reliable Broadcast” by Coan et al. discusses a DBRB protocol that uses a bulletin board to facilitate the broadcast of messages in a large-scale distributed system. The protocol is designed to be efficient and scalable, and can tolerate up to one third of the nodes being Byzantine without affecting the reliability of the broadcast.
The solutions of the two papers that we have seen are more robust to failures, but they have not very high performance in the reality, while the solution using a central server is more efficient but less robust in overall, what we want to achieve is a trade-off of the two solution. Because the problem is very important are needed more efficient solution, but it is very difficult to maintain in the same time the robustness.