YACA: Yet Another Chat Application

Group 23: Simon Egger

1 Introduction

YACA is *yet another chat application*[1] aiming to cover the project requirements: Dynamic discovery, leader election, reliable ordered multicast, and Byzantine fault tolerance.

YACA uses the client-server architecture to enable clients to participate in one global chat room, where clients are able to respond to previous messages (i.e. the happened-before relation is respected). In general, our strategy is to utilize powerful building blocks like *total-ordered reliable multicast*, *view-synchronous group communication*, and the *(Max-)Phase-King Algorithm* to realize this project in a Byzantine fault-tolerant manner.

This report starts with an overview in Section 2, how our project is able to fulfill the project requirements. Afterwards, Section 3 provides a high-level description of our architecture design, whereupon Section 4 provides a more detailed description of our building blocks. We conclude in Section 5. Furthermore, we design the *Max-Phase-King Algorithm*, where Appendix A provides additional details and proofs of correctness.

2 Project Requirement Analysis

1. *Dynamic Discovery:* We realize view-synchronous group communication (cf. Section 4.6). Servers use this mechanism to ensure they manage a consistent group-view, which is for example required for TO-R-multicast and the (max-)phase-king algorithm.
2. *Leader Election:* We use the more powerful phase-king algorithm to realize our election algorithm (cf. Section 4.4). We elect a *Manager (MGR)* among the servers, which is responsible for listening to new join requests via broadcast.
3. *Reliable Ordered Multicast:* We realize causal- and total-ordered reliable multicast (cf. Section 4.1, Section 4.5). We use CO-R-multicast for sending group chat messages, executing the election and phase-king algorithm, and realizing the hold-mechanism of our view-synchronous group communication implementation. We use TO-R-multicast to deliver election announcements and join messages of new servers to ensure the *agreement* and *order* guarantee of view-synchronous group communication.
4. *Fault Tolerance:* We aim to provide Byzantine fault-tolerance. We describe in Section 4 how the individual building blocks of our implementation are realized to ensure Byzantine fault-tolerance. Furthermore, we use active replication to replicate the group chat messages.

[1] `https://github.com/eggersn/YACA`

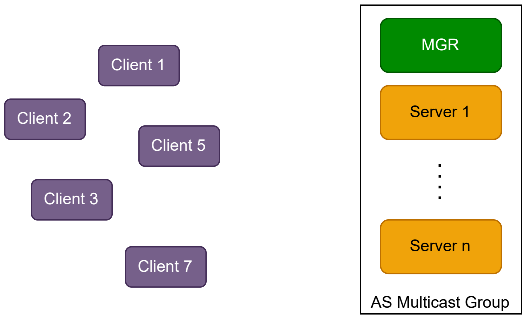

Fig. 1. Overview of the System Architecture

3 Architecture Design

An overview of the YACA system architecture can be found in Figure 1. YACA uses the client-server architecture, where clients are participants of a (single) global group chat and servers ensure that the chat history is replicated in a Byzantine fault-tolerant manner. In the following, we provide an overview of available administrator configuration, along with basic server and client functionality:

3.1 Administrator Configuration

Before spawning system-components, the administrator is able to configure multiple components that remain unchanged during execution. This comprises the following settings:

- Class D Internet address: An address, preferably within the Local Network Control Block (224.0.0.0 to 224.0.0.225), used by the servers for multicast traffic.
- 5 × UDP ports: Four predefined ports used by servers to consume new multicast messages (cf. Section 3.2) and one predefined port for listening to broadcast messages.
- Timeout settings: Specification for different timeouts that might occur (e.g. heartbeats, polling-rates, timeouts for crash detection)
- Initial view: Specification of servers that are spawned at first startup. Each server is specified with an identifier, an IP address and listening-port, and a public key for digital signatures.[2]

3.2 Server and Clients

On startup, the server is either part of the initial view, or joins the group by using the join-protocol described in Section 4.6. Most importantly, we use view-

[2] Note that the initial view is solely used by the servers as a base case, allowing new dynamically spawned servers the build a web of trust (cf. Section 4.6).

synchronous group communication, entailing that the new server builds a group view that is consistent with other correct servers' group views. Therefore, using a combination of view-synchronous group communication and digital signatures, we can realize closed group communication. As described in Section 3.1, we use one multicast address where servers consume multicast messages over four UDP ports, each serving a different purpose:

i) C-Read: R-multicast (cf. Section 4.1), where clients can query heartbeats and public keys of other clients.
ii) C-Write: CO-R-multicast (cf. Section 4.1), where clients send new group chat messages to, or register to the service.
iii) S-Coordination: CO-R-multicast, where *exclusively* servers can perform the election algorithm and the phase-king algorithm.
iv) S-Announcements: TO-R-multicast (cf. Section 4.5), where *exclusively* servers announce elections and information about new servers is announced.

C-Read Clients send new group chat messages to C-Write. To reduce the load of the servers, clients are only allowed to send messages, but not heartbeats or negative acknowledgements (cf. Section 4.1). Therefore, clients can periodically query servers' heartbeats via C-Read. Note that this mechanism can also be used by the clients to detect if there were omission failures. In response, the client receives heartbeats from multiple servers, which can then be contracted individually, via unicast, to retrieve the new group chat messages. Since the retrieved group chat messages contain vector timestamps, the client is able to recreate the chat history while respecting the happened-before relation.

The second important usage of C-Read is that a client can query public keys of other clients: Since clients sign every group chat message they sent, including the vector timestamp, malicious (i.e. Byzantine fault) servers cannot tamper with group chat messages, once the client knows about the corresponding public keys. To gain knowledge about these public signing keys, the client multicasts a query to the group of servers, collects their responses and performs a majority decision.

C-Write On startup, clients create a random user ID and a digital signature key-pair. They then sent both via C-Write to the servers, which perform the phase-king algorithm (cf. Section 4.3) on the received data to ensure consensus.[3] The servers respond afterward to notify the client whether the registration was successful. If necessary, the client retries the registration process.

Additionally, clients send their new group chat messages to C-Write. To ensure that other clients are able to rebuild the chat history while respecting the happened-before relation, new group messages contain the client's vector timestamps. Furthermore, the message, including the vector timestamp, is signed by the client to ensure that malicious servers cannot tamper with it.

[3] This is required in case the client is malicious or there are multiple clients with the same user ID registering at the same time.

S-Coordination Since TO-R-multicast used by S-Announcements has a non-negligible overhead, we use S-Coordination for messages that do not have to be total-ordered. There are mainly two applications for S-Coordination: When delivering a message via S-Announcements, we perform the phase-king algorithm on the delivered message contents to ensure that every correct server processes the same message.[4] The second application is that S-Coordination can be used to perform elections, after delivering election announcements via S-Announcements.

S-Announcements As previously stated, S-Announcements is used to deliver election announcements and to announce newly joined servers. This ensures that join messages, as part of view-synchronous group communication, are delivered in the same order (cf. Section 4.6), and that every server has the same group-view when starting an election.

Note that the phase-king algorithm also requires a consistent group-view, analogously to the election algorithm. Since the phase-king algorithm is also used for C-Write, while the max-phase-king algorithm is used to realize TO-R-multicast itself (cf. Section 4.5), we terminate all executions of the phase-king algorithm when S-Announcements delivers a join message.[5]

3.3 Manager (MGR)

We use leader election to determine a unique *Manager (MGR)* of the servers. While the MGR behaves, for the most part, identical to other servers, he also listens to incoming broadcast messages. This is used by new servers to contact a server instance who is able to send messages to S-Announcements. This is required since servers only accept messages over S-Announcement, which were sent by other participants they already know.[6]

[4] This is required, since the original sender might send different messages, with the same sequence number, to different servers.

[5] To be more exact, we use hold-messages as described in Section 4.6 to ensure that either every or no server terminates a phase-king execution.

[6] For the sake of full transparency, we note that this utilization of the MGR is neither the most practical nor the most efficient one, and is instead mostly used to fulfill the formalities of the project requirements. For example, one disadvantage of this approach is that other servers would have to constantly monitor the MGR, by listening to broadcast themselves, in order to ensure that the *non-triviality* guarantee of view-synchronous group communication is satisfied, since the MGR himself might be malicious. At the time of writing this report, this monitoring feature was not yet implemented.

4 Implementation Details

4.1 Primitives

Notation We use V to denote the (finite) set of server processes. To coincide with the notation of the lecture and [2], we sometimes use g do denote the server group.

Messages In general, all messages inherit the base class Message, which includes fields for the message Header, Content, and Metadata. Furthermore, base functionalities include encoding and decoding the messages, using JSON. The message type is uniquely identified by the header, which tells the process how the message's content should be processed and which metadata is included. Examples for included metadata are digital signatures, sequence numbers, piggybacked acknowledgements, or nonces.

Delivery Queues Another useful primitive is the FIFO delivery queue. Our general approach for processing incoming messages is to have one listening thread per listening port, which handles configuration messages (e.g. heartbeat messages, negative acknowledgements, or proposal messages for TO-multicast) and forwards only the relevant messages via a delivery queue to one or multiple worker threads. Therefore, we use locks to realize this in a thread-safe manner. Furthermore, we use semaphores to provide a blocking consume and a non-blocking produce operation.

Reliable Multicast (R-Multicast) In general, we implement R-multicast using a combination of reliable multicast over IP multicast, as described in [2, p. 649]. Each process $p \in V$ manages a local sequence number S_p, which is initialized as zero and is atomically incremented each time p sends a message, and a dictionary $R_g : V \to \mathbb{N}$, where $R_g(q)$ keeps track of q's last delivered message sequence number (for each $q \in V$). When sending a message (via IP multicast), p includes both S_p and $R_g^p := \{(q, R_g(q)) | q \in V\}$ as metadata. Therefore, when a message m is received with the attached sequence number S by some process q, there are three possible cases:

i) $S \leq R_g(q)$: p already delivered m and simply discards the duplicate
ii) $S = R_g(q) + 1$: p delivers m and additionally stores it indefinitely
iii) $S > R_g(q) + 1$: p detects that there are missing messages and places m in a holdback-queue

Note that piggybacking R_g^p to a message implicitly acts as an acknowledgement. As this only works if p continuously sends messages, we periodically send heartbeat messages as well.

If p detects missing messages, by inspecting piggybacked acknowledgements of normal messages or heartbeat messages, it sends negative acknowledgements to the sender (which is not necessarily the original sender of the message). Since each message is digitally signed, we assume that messages forwarded in this manner cannot be manipulated by the forwarding process (which is an additional guarantee on top of the checksums used internally for IP multicast). After missing messages are delivered, p checks its holdback-queue for messages that are now able to be delivered.

Similar to the original protocol described in [2, p. 649], it is easy to see that above protocol satisfies the following properties:

- *Integrity:* Duplicates are detected by inspecting the attached sequence number. IP multicast uses checksums, and we use additional digital signatures to ensure that the received message is identical to the message signed by the sending process.
- *Validity:* Trivially satisfied by the usage of IP multicast.
- *Agreement:* Each process is able to detect that some process q delivered m by inspecting R_g^q attached to the following messages. Since q stores m indefinitely, p can request the missing message by sending negative acknowledgements.

Causal-Ordered Reliable Multicast (CO-R-Multicast) The protocol above can now be extended to support causal ordering by using the strategy described in [2, p. 657-658]: Each process manages a further holdback-queue and an additional vector stamp $R_g^{CO} : V \to \mathbb{N}$, where $R_g^{CO}(q)$ manages the last CO-R-delivered message sequence number of process q. When process p R-delivers a message m of sender q with attached vector stamp R and sequence number S, it places m in the second holdback-queue until the contraints

i) $R_g^{CO}(q) + 1 = S$, and
ii) $R_g^{CO}(q') \geq R(q')$ for each $q' \in V \backslash \{q\}$

are satisfied. The proof of correctness is identical to the one given in [2, p. 658].

4.2 Suspicion Protocol

When having malicious servers in our system it is advantageous to have a protocol that is able to suspend them, if inconsistent or incorrect behavior is detected. In this context, suspending a server means that no more incoming messages are accepted anymore, or to be more exact, no new message is accepted from the suspended server that was not already acknowledged by another server. This can easily be achieved by using digital signatures and only accepting messages that are gathered via negative acknowledgements.

One straight-forward strategy to achieve an agreed suspension is to send *suspect messages* to all other servers and suspend a server as soon as at least one suspect message has arrived. To avoid having a malicious server sending suspect messages for all servers and suspending them, we modify this protocol to allow for $f < n/3$ malicious servers (since we utilize the phase-king algorithm this limit is of course capped by $f < n/4$):

- If detecting inconsistent or incorrect behavior (e.g. timeout on response), send a suspect message. The suspect message contains the suspected server identification and the reason why the server is suspected (e.g. ”proposal timeout on *msg_id*” for the ISIS algorithm).
- If a server gathers more than f suspect messages for the same server and the same reason, he sends a suspect message himself if he did not already send one.
- If a server gathers at least $n - f$ suspect messages, the server is suspended (cf. Section 4.6).

For the guarantees of the above protocol, it is easy to see that if a server gathers $n - f$ suspect messages, he can be sure that all other correct processes gather at least $n - 2f > f$ (for $f < n/3$) suspect messages of the same kind. Thus, eventually, every correct process eventually sends a suspect message. On the other hand, if a server receives at least f suspect messages, he can be sure that at least one correct (or honest) server is suspecting the accused server.

4.3 (Max-)Phase-King Algorithm

To provide resilience against Byzantine faults (with $f < n/4$), we use the Phase-King Algorithm [1] and our Max-Phase-King Algorithm (cf. Appendix A.1). We use CO-R-multicast which separates different phases and their rounds more elegantly, i.e., the phase-king message of round 2 is delivered after all round 1 messages are delivered.

Another important aspect is that each participant agrees on the same phase-king. The same problem occurs for our election mechanism and is solved by utilizing view-synchronous group communication (cf. Section 4.6). This allows us to manage a consistent list of unsuspended server IDs across all server instances, yielding that the ith entry of the list is selected as the phase-king of phase i.

Furthermore, we use the suspicion protocol described in Section 4.2. This ensures that processes are not stuck waiting for messages in round 1, when gathering N messages, or in round 2, when receiving the tiebreaker value of the phase-king. In case the phase-king is suspended, the next phase-king simply takes over (avoiding the need to choose the default value as the tiebreaker).

4.4 Election

It is easy to see that ring-based election algorithms are prone to a wide range of failures, making them unappealing. Analogously, the bully algorithm [2, p. 644-645] does not provide resilience against Byzantine failures, e.g., an arbitrary processes might immediately send *coordinator* messages, causing an inconsistent result for the election. Therefore, we realize a Byzantine fault-tolerant election algorithm by utilizing more powerful building blocks: The phase-king algorithm (cf. Section 4.3) and group-synchronous group communication (cf. Section 4.6).

```
function Election(servers : list) for $p_i \in V$ begin
```

```
2      elected = $\perp$
3      for server in servers do
4          init_value : $\{0,1\}$
5          if is_reachable(server) then
6              init_value $\leftarrow$ 1
7          else
8              init_value $\leftarrow$ 0
9
10         result $\leftarrow$ phase_king(init_value)
11         if result == 1 then
12             elected $\leftarrow$ server
13             return
14     done
15 end
```

It is easy to see that above election algorithm satisfies both *safety* and *liveness* requirements. We note that we use the definitions of the lecture, which slightly deviate from the ones given in [2, p. 642], and adopt the notation: Let V be the set of processes, and $V_g \subseteq V$ the set of correct processes. Each process $p_i \in V$ has a variable $elected_i$, which is either undefined (i.e. $elected_i = \perp$) or contains the identifier of the elected process. Safety and liveness are then defined as:

i) *Safety:* $\forall p_i \in V_g : elected_i = \perp$ or $(\forall p_j \in V_g : elected_i = elected_j)$
ii) *Liveness:* Every correct process $p_i \in V_g$ eventually sets $elected_i \neq \perp$

For safety, it is easy to see that if $p_i \in V_g$ sets $elected_i = p_k \neq \perp$ for some $p_k \in V$, the result of the phase-king algorithm is 1 for p_k. Therefore, every other correct process p_j has the same result for the phase-king algorithm,[7] resulting in $elected_j = p_k$. Note that this implicitly requires that every process shares the same input list `servers`, which we guarantee be using view-synchronous group communication.

For liveness, we use the fact that the phase-king algorithm requires $f < n/4$. Therefore, the for-loop in line (3) eventually selects a correct server p_k, i.e., after at most $f + 1$ iterations. By assumption, the p_k responds to the incoming ping-messages of other servers, resulting in $init\_value_i = 1$ for every $p_i \in V_g$. Since $|V_g| = n - f > n/2 + f$, the correct processes always use the majority value in round 2 of every phase in the phase-king algorithm. Therefore, the result of the phase-king algorithm is 1 as well and $elected_i$ is set to $p_k \neq \perp$.[8]

4.5 Total-Ordered Reliable Multicast (TO-R-Multicast)

We build TO-R-multicast on top of CO-R-multicast using the ISIS algorithm described in [2, p. 655-656]. We use CO-R-multicast instead of R-multicast for the following three reasons:

[7] The proof of this guarantee is similar to the one presented in Theorem 1

[8] A similar argument provides an even stronger guarantee than safety, where the elected process appears as ”not-crashed” to at least one correct process.

i) Proposal messages are delivered after the original message
ii) To implement the suspicion protocol (cf. Section 4.2) and to suspend processes (cf. Section 4.6)
iii) To perform the max-phase-king algorithm (cf. Appendix A.1) on the agreed sequence numbers

We note that this, however, does not guarantee that the total-ordered messages satisfy causal-ordering as well.

Each server p maintains variables P_g^p and A_g^p. Upon sending a message via TO-R-multicast, the sender adds a randomly generated message ID and CO-R-multicasts it. When a server CO-R-delivers such a message, he responds with a proposal $P_g^p = max(A_g^p, P_g^p) + 1$ for the message's sequence number. This is also done via CO-R-multicast, as opposed to unicast [2, p. 655-656]. This allows us to use the max-phase-king algorithm directly as described in Appendix A.1. While waiting for all proposals to arrive, the message is stored in a sorted hold-back queue. For sorting, we use the tuples $(seqno, msg\_id)$, where $seqno$ is the largest received proposal and

$$\begin{aligned}(seqno1, msg\_id1) < (seqno2, msg\_id2) \iff &(seqno1 < seqno2 \text{ or} \\ &(seqno1 = seqno2 \text{ and } msg\_id1 < msg\_id2)).\end{aligned}$$

When the max-phase-king algorithm completed its execution for a message (v_{init} is the largest gathered proposal), its respective entry in the hold-back queue is marked as *ready-to-deliver*. A message is moved from the hold-back queue to the FIFO-ordered delivery queue if and only if it is marked as ready-to-deliver, and it is first in the hold-back queue. As before, we use the suspicion protocol (cf. Section 4.2) if processes experience timeouts while waiting for proposals.

For the proof of correctness, consider two correct processes p_i and p_j, where p_i delivers a message msg_1 before msg_2. Since we use a FIFO-ordered delivery queue, msg_1 is added to the queue before msg_2. Consider the time-point, where p_i adds msg_1 to the delivery queue. There are two cases to inspect:

i) *p_i did not propose a sequence number for msg_2 yet:* Let S_1 be the assigned sequence number of msg_1 (after the max-phase-king algorithm). Then, we have $S_1 \leq A_g^{p_i}$ and p_i proposes some sequence number P_2 for msg_2 with $P_2 = max(A_g^{p_i}, P_g^{p_i}) + 1 > A_g^{p_i} \geq S_1$. Since p_i and p_j are correct, p_j shares the same value for S_1 and the max-phase-king algorithm further ensures that the final sequence number S_2 for msg_2 is larger than S_1 (cf. Theorem 1). Therefore, p_j also delivers msg_1 before msg_2.
ii) *p_i did already propose a sequence number for msg_2:* Using the same notation as above, we know that msg_1 is first in the ordered hold-back queue, before it is appended to the delivery queue. Let P_2 be the largest proposal for msg_2 that was received by p_i yet. Since $(S_1, id(msg_1)) < (P_2, id(msg_2))$, we either have $S_1 < P_2$, or $S_1 = P_2$ and $id(msg_1) < id(msg_2)$. In the first case, the max-phase-king algorithm is later executed for msg_2 with $v_{init}^i \geq P_2$, resulting in an output $S_2 \geq P_2 > S_1$ (cf. Theorem 1). In the second case, we have the same argument resulting in $S_2 \geq P_2 \geq S_1$, where

$id(msg_1) < id(msg_2)$ again decides that msg_1 is delivered before msg_2. Therefore, it can be seen that p_j delivers msg_1 before msg_2 as well.

4.6 View-Synchronous Group Communication

In view-synchronous group communication, new processes can join and participants can be suspended.

i) *Suspension:*[9] We extend the suspicion protocol described in Section 4.2. Furthermore, our servers listen to multiple multicast instances (i.e. one for TO-R-multicast and two for CO-R-multicast), which is also considered in the following:
   - If a server p detects inconsistent or incorrect behavior, or receives more than f suspect messages (cf. Section 4.2), p sends a suspect message via each multicast instance.
   - If p gathers at least $n - f$ suspect messages for servers $q_1, \ldots, q_k$, he sends *hold-messages* $h_1, \ldots, h_k$, where h_i contains information about q_i, and stops sending new messages, except for additional suspect and hold messages. He waits for the corresponding hold messages for $q_1, \ldots, q_k$ of all other servers. If a server q does not send a hold message within a timeout interval, p send a suspect message for q as well.
   - Let $q_1, \ldots, q_k, \ldots, q_{k+l}$ be the enqueued servers for suspension (i.e. p received more than $n - f$ suspect messages for each q_i). After p received $n - k - l$ different hold messages for each q_i on each multicast instance, he suspends every q_i and continues normal operation.

ii) *Join:* On first startup, the administrator can configure an *initial view*, where multiple server instances are spawned and share a consistent group view. This initial view contains information about the server ID, IP address and listening-port for unicast messages, and a public key for digital signatures. Servers that are spawned afterwards also know the initial view, but yet have to learn about the servers that joined afterwards (e.g. all servers included in the initial view might already have crashed).
   - When a new server p wants to join the group of servers, he sends a request to the MGR (via broadcast) containing p's ID, IP address and listening-port, and a public key for digital signatures.
   - When the MGR receives such a request he validates that p's ID is unique, encapsulates the request in a signed *join-message*, and forwards it via TO-R-multicast. Note that signing the join-message is crucial as other servers only accept messages of participants. Furthermore, note that other servers can validate the correct behavior of the MGR by listening to broadcast as well.

[9] For the sake of full transparency, we note that, at the time of writing, this feature is not implemented yet. Instead, we suspend a server directly after receiving $n - f$ suspect messages as described in Section 4.2. For a single CO-R-multicast instance, this approach is sufficient since both phase-king algorithm and ISIS algorithm are highly synchronized. In contrast to this, there remains a small probability that, for multiple CO-R-instances, this causes race-conditions and leads to inconsistent behavior.

- When a participant q delivers the join-message of p, q sends an acknowledgement to p adds the encapsulated information to its group-view, but marks p as still *inactive*. Furthermore, q sends a hold-message on each multicast instance (identical to i)).
- When p receives the first acknowledgement, he is now able to send negative acknowledgements and is therefore able to consume the messages of every multicast instance. When reaching the corresponding hold-messages (containing p's ID), p sends a *commence-message* to tell other participants to resume normal operation.[10]
- When q receives a commence-message from p on each multicast instance, he marks p as *active* and continues normal operation. Note that the first message of p is expected to be a commence-message. Otherwise, q simply sends a suspect message for p. Similarly, q is able to set a timeout interval.

When considering view-synchronous group communication as defined in [2, p. 772-773], we need to argue that the guarantees *order, integrity, non-triviality, agreement, and validity* are satisfied. Due to space restrictions, we include the proof of correctness in Appendix A.2.

5 Conclusion and Discussion

In this report, we present *Yet Another Chat Application (YACA)*, which allows multiple clients to chat in one global chat-room. Chat messages are replicated using *active replication*, making clients resilient against malicious servers. On server-side, we use view-synchronous group communication, reliable total-ordered multicast, and the (max-)phase-king algorithm as powerful building blocks to reach consensus and to maintain a consistent group-view. For each building block, we argue in Section 4 how our implementation achieves high resilience against Byzantine faults. In particular, we design the max-phase-king algorithm (cf. Appendix A.1) specifically for reaching consensus on the sequence numbers used by the ISIS algorithm, while also ensuring that the output value is larger than every input value of the correct processes.

For the sake of full transparency, we note that, at the time of writing, there are two parts respected in this report which are not implemented yet (cf. Footnotes 6, 9)

TODO Discussion

[10] Most importantly, this mechanism allows p to build its own consistent group view: Using an inductive argument, p starts with the initial view (i.e. the base case) and is able to verify the signatures of the initial processes. Any process q that joined thereafter must have used the same protocol as p, which allows p to validate that q was indeed added to the group-view. This way, p also learns of q's public key, which enables p to build a web of trust and validate new messages of q. Analogously, p collects suspect messages and suspends processes if necessary. By simulating previous elections and phase-king executions, p is then able to rebuild the consistent state and group-view in a Byzantine fault-tolerant manner.

References

1. Piotr Berman and Juan A. Garay. Cloture votes:n/4-resilient distributed consensus in t + 1 rounds. *Theory of Computing Systems*, 26(1):3–19, March 1993.
2. George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011.

A Appendix

A.1 Max-Phase-King Algorithm

```
function MaxPhaseKing($v_{init}$: int, $f$: int) for $p_i \in V$ begin
    // Preparation
    broadcast $v_{init}$ to all processes
    await value $v_j$ from each process $p_j \in V$
    $\tilde{v}$ ← $max\{v_j \mid p_j \in V\}$
    $v$ ← $\tilde{v}$

    for $phase$ = $1, \ldots, f+1$ do
        // Round 1
        broadcast $v$ to all processes
        await value $v_j$ from each process $p_j \in V$
        $mode$ ← mode of the $v_j$'s
        $mult$ ← number of times that $mode$ occurs
        $median$ ← median of the $v_j$'s

        // Round 2
        if $i = phase$ then
            broadcast $\lceil median \rceil$ to all processes
        receive $tiebreaker$ from $p_{phase}$
        if $mult > |V|/2 + f$ then
            $v$ ← $mode$
        else if $|\{v_j \mid v_j \leq tiebreaker\}| > f$
            $v$ ← $tiebreaker$
    done

    output $v$
end
```

Notation As before, we use V to denote the set of all processes. Let $n := |V|$ and $f < \lceil n/4 \rceil$ be the number of malicious processes. We denote $V_g \subseteq V$ as the set of correct processes and use the notation $v^{(p,k)}$ to denote the value of v for process p in phase k ($k = 0$ is used for the preparation phase). If the variable does not change with different phases, we omit k.

Proof of Correctness Our main goal is to show that all correct processes output the same value v and that $v \geq v_{init}^{(p)}$ for all $p \in V_g$.

Lemma 1. *Let p_i be the phase-king. If p_i is correct then $|\{v_j^{(p_k,i)} \mid v_j^{(p_k,i)} \leq tiebreaker^{(p_k,i)}\}| > f$ for all correct processes $p_k \in V_g$.*

Proof. If p_i is correct, he gathers $\{v_j^{(p_i,i)} \mid p_j \in V\}$. After rewriting these values as

$$v_{(1)} \leq v_{(2)} \leq \ldots \leq v_{(n)},$$

the median is computed as $median = v_{(\frac{n+1}{2})}$ if n is odd, or $median = (v_{(\frac{n}{2})} + v_{(\frac{n}{2}+1)})/2$ if n is even. In both cases we have that

$$|\{v_j^{(p_i,i)} \mid v_j^{(p_i,i)} \leq \lceil median \rceil\}| \geq \left\lceil \frac{n}{2} \right\rceil,$$

which in turn yields:

$$\begin{aligned} |\{v_j^{(p_k,i)} \mid v_j^{(p_k,i)} \leq tiebreaker^{(p_k,i)}\}| &\geq |\{v_j^{(p_i,i)} \mid v_j^{(p_i,i)} \leq \lceil median \rceil\}| - f \\ &\geq \left\lceil \frac{n}{2} \right\rceil - f \\ &> \left\lceil \frac{n}{2} \right\rceil - \left\lceil \frac{n}{4} \right\rceil \\ &\geq \left\lceil \frac{n}{4} \right\rceil - 1 \qquad \geq f \end{aligned}$$

□

Lemma 2. *Let p_i be the phase-king. If p_i is correct, then all correct processes share the same value for v after the current phase.*

Proof. Let p_j and p_k be arbitrary correct processes. Due to Lemma 1, v is updated with the *mode* or the *tiebreaker* value. Therefore, there are three cases to consider:

i) p_j and p_k use their *mode* value: Then, p_j received $mode^{(p_j,i)}$ more than $n/2 + f$ times, which implies that p_k also received $mode^{(p_j,i)}$ more than $n/2$ times. Therefore, there is no other possibility than $mode^{(p_k,i)} = mode^{(p_j,i)}$.

ii) p_j uses its *mode* value and p_k uses the *tiebreaker* value: As before, p_j received $mode^{(p_j,i)}$ more than $n/2+f$ times, which implies that the phase-king p_i also received $mode^{(p_j,i)}$ more than $n/2$ times. Therefore, we have $median^{(p_i,i)} = mode^{(p_j,i)} \in \mathbb{N}$, which is then received by p_k as the *tiebreaker*.

iii) p_j and p_k use the *tiebreaker* value: As p_i is assumed to be correct, both receive the same *tiebreaker* value.

□

Lemma 3. *Let $1 \leq i \leq f + 1$ be a phase with a correct phase-king. Then, the following holds true:*

$$\forall p_j, p_k \in V_g \forall phase \in \{i, \ldots, f+1\} : v^{(p_j,phase)} = v^{(p_k,phase)}$$

That is, once there was a correct phase-king, all correct processes continue to share the same value for v afterwards.

Proof. Lemma 2 handles the case of $\mathit{phase} = i$. Afterwards, all correct processes broadcast the same value of v, which implies that $\mathit{mult}^{(p_j,\mathit{phase})} \geq n - f > n/2 + f$ is satisfied for all $p_j \in V_g$ and $i < \mathit{phase} \leq f + 1$. Using the same argument as i) in the proof of Lemma 2, it is easy to see that the statement holds true. □

Lemma 4. *The following statement holds true:*

$$\forall p_j, p_k \in V_g \forall \mathit{phase} \in \{0, \dots, f + 1\} : v^{(p_j,\mathit{phase})} \geq v_{\mathit{init}}^{(p_k)}$$

That is, the value of v for a correct process p_j is always larger than the initial values of all correct processes.

Proof. We use an inductive argument to prove this statement. For $\mathit{phase} = 0$, $v^{(p_j,0)}$ is initialized in line 6. As all correct processes p_k broadcast their initial value $v_{\mathit{init}}^{(p_k)}$ the statement is trivially satisfied.

For $\mathit{phase} > 0$, $v^{(p_j,\mathit{phase})}$ there are three cases to consider:

i) $v^{(p_j,\mathit{phase})}$ is not updated, i.e., $v^{(p_j,\mathit{phase})} = v^{(p_j,\mathit{phase}-1)}$: By the induction hypothesis, the statement holds true.

ii) $v^{(p_j,\mathit{phase})}$ is updated in line 21. Then $\mathit{mult}^{(p_j,\mathit{phase})} > n/2 + f$ which implies that at least one correct process p_l sent $\mathit{mode}^{(p_j,\mathit{phase})}$ at the beginning of the phase. Using the induction hypothesis, we have:

$$v^{(p_j,\mathit{phase})} = \mathit{mode}^{(p_j,\mathit{phase})} = v^{(p_l,\mathit{phase}-1)} \geq v_{\mathit{init}}^{(p_k)}, \quad \forall p_k \in V_g$$

iii) $v^{(p_j,\mathit{phase})}$ is updated in line 23. Let

$$C = \{v_k^{(p_j,\mathit{phase})} \mid v_k^{(p_j,\mathit{phase})} \leq \mathit{tiebreaker}^{(p_j,\mathit{phase})}\},$$

then the condition in line 22 states $|C| > f$. Therefore, there exists at least one correct process p_l with $v_l^{(p_j,\mathit{phase})} \in C$. This yields:

$$v^{(p_j,\mathit{phase})} = \mathit{tiebreaker}^{(p_j,\mathit{phase})} \geq v_l^{(p_j,\mathit{phase})} = v^{(p_l,\mathit{phase}-1)} \geq v_{\mathit{init}}^{(p_k)},$$

for all $p_k \in V_g$.

□

Theorem 1. *Each correct process p_j that executes the max-phase-king algorithm outputs $v^{(p_j,f+1)}$ that satisfies*

i) $\forall p_k \in V_g : v^{(p_j,f+1)} = v^{(p_k,f+1)}$, *and*

ii) $\forall p_k \in V_g : v^{(p_j,f+1)} \geq v_{\mathit{init}}^{(p_k)}$.

Proof. Lemma 3 proves statement i) and Lemma 4 proves statement ii). □

A.2 View-Synchronous Group Communication

For formal reasons, we consider a sequence of views

$$v_0(g), v_1(g), \dots, v_i(g), \dots,$$

where $v_0(g) = (p_1, \dots, p_l)$ is the initial view specified by the administrator and $v_i(g)$ is of the following form:

$$\begin{aligned} v_i(g) &= v_{i-1}(g) + (p) - (q_1, \dots, q_l), && \text{or} \\ v_i(g) &= v_{i-1}(g) - (q_1, \dots, q_l), && \forall j \in \{1, \dots, l\} : q_j \in v_{i-1}(g). \end{aligned} \tag{1}$$

That is, a view delivery includes at most one joining process and potentially multiple suspended processes.

In the following, we show that our implementation described in Section 4.6 satisfies the guarantees described in [2, p. 772-773]. As before, we assume a maximum of $f < n/4$ incorrect processes. Therefore, we restrict the *non-triviality* guarantee to group partitions, where one partition contains more than $2n/3$ processes and demand only that this partition continues to function.

i) *Order:* Let p and q be correct processes that deliver the views $v(g)$ and $v'(g)$. If p delivers view $v(g)$ before $v'(g)$, there exists $i, j \in \mathbb{N}_{\geq 1}$ with $v(g) = v_i(g)$ and $v'(g) = v_{i+j}$. Due to the total-ordering of join-messages, we can also assume that $v_i(g), \dots, v_{i+j}(g)$ are of the second form, as described in Equation (1) (otherwise, *Order* is trivially satisfied). Before p delivers $v(g)$, p collects hold-messages from all processes that are not enqueued for suspension. In particular, p received corresponding hold messages from q, which entails that q was also collecting suspect messages for the same processes as p. Using the guarantees of our suspicion protocol (cf. Section 4.2), it is easy to see that p and q eventually suspend the same processes. Therefore, q also delivers $v(g)$ before $v'(g)$.

ii) *Agreement:* In addition to the *Order* guarantee, *Agreement* also demands that every correct process delivers the same messages in every view. This is achieved by using hold-messages as described in Section 4.6 in combination with CO-R-multicast.

iii) *Integrity:* If a correct process p delivers view $v(g)$, then p is not suspended since it would have terminated otherwise. Therefore, $p \in v(g)$ holds true. Furthermore, if p delivers a message m, then the guarantees of CO-R-multicast entail that p will not deliver m again. Finally, the usage of hold-messages ensures that *sender*(m) is in the view in which p delivers m.

iv) *Non-triviality:* If a correct process q joins the group, it contacts the MGR via broadcast to forward the join-message to all participants via TO-R-multicast. As other participants also listen to broadcast, they can theoretically[11] detect if the MGR does not forward join requests and send suspect messages. Therefore, we can assume that the join-message of q is eventually delivered

[11] For the sake of full transparency, we note that we did not implement this feature yet.

by every participant via TO-R-multicast. By construction, q then consumes the messages of every multicast instance and is afterwards considered as an active participant.

Analogously, correct processes eventually detect group partitions via timeouts and send suspect messages. Using the guarantees of our suspicion protocol (cf. Section 4.2), the processes of other partitions are eventually suspended, as long as the own partition contains more than $2n/3$ processes.

v) *Validity:* Because of the validity guarantee of CO-R-multicast, correct processes always deliver the messages that they send. If the system fails to deliver a message to any process q, other correct processes will notice a timeout when waiting for the hold-messages of q before delivering a new view (if the timeout is not noticed before). Since this timeout is detected by $n-f$ correct processes, q will be suspended and excluded from the next view.