Hydra - Companion document

Author(s):
Biri, Andreas; Da Forno, Reto; Kuonen, Tobias; Mager, Fabian; Zimmerling, Marco; Thiele, Lothar

Publication date:
2023-03-06

Permanent link:
https://doi.org/10.3929/ethz-b-000602742

Hydra - Companion document

Andreas Biri, Reto Da Forno, Tobias Kuonen, Fabian Mager,
Marco Zimmerling, Lothar Thiele

06.03.2023

The primary publication of Hydra is our conference publication "Hydra: Concurrent Coordination for Fault-tolerant Networking" at IPSN '23:

```
@inproceedings{biri2023hydra,
  author = {Biri, Andreas and Da Forno, Reto and Kuonen, Tobias and Mager,
Fabian and Zimmerling, Marco and Thiele, Lothar},
  title = {Hydra: Concurrent Coordination for Fault-tolerant Networking},
  year = {2023},
  isbn = {9798400701184},
  publisher = {Association for Computing Machinery},
  address = {New York, NY, USA},
  url = {https://doi.org/10.1145/3583120.3587047},
  doi = {10.1145/3583120.3587047},
  booktitle = {The 22nd International Conference on Information Processing in
Sensor Networks},
  pages = {},
  numpages = {14},
  location = {San Antonio, TX},
  series = {IPSN '23}
}
```

This companion document to our conference publication contains the following additions:

1. Section 1 briefly re-introduces the notation and requirement definitions we will use for the rest of this document.
2. Section 2 presents the bootstrapping algorithm for Hydra.
3. Section 3 presents an extended version of Hydra's algorithm as it has been implemented in software and contains a more detailed explanation of its components, including details on the technical implementation of Hydra.
4. Section 4 contains the full formal proofs of Hydra.

1 Introduction

Overview. We have a network $\mathcal{N}$ of N nodes i with $\mathcal{N} = \{i \in \mathbb{Z}_{>0} \mid 1 \leq i \leq N\}$ which communicate in rounds with period T, where F rounds form an epoch. A communication round consists of three phases as shown in Figure 1, namely `DD` (data dissemination), `SN` (schedule negotiation), and `SD` (schedule distribution). In the `DD` phase, one node per slot can flood a packet containing application data. In the `SN` phase, nodes concurrently receive, process, and

send packets, with the goal of reaching a steady state across the network where each node possesses the same packet. Reaching this steady state is not guaranteed as nodes may send in any order without delivery guarantees. In the SD phase, one or more nodes flood the same scheduling packet at the same time.

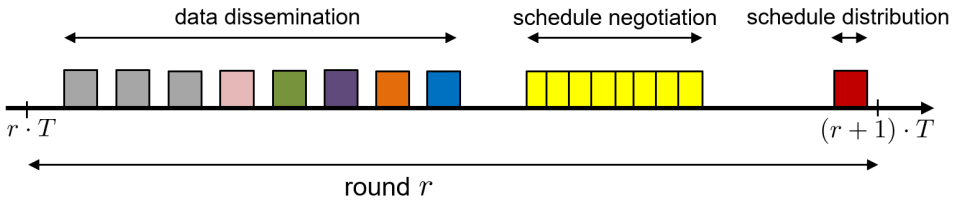

Figure 1: Round structure of Hydra for the r-th round.

Notation. We will make use of the symbol $\perp$ to denote that a variable contains no information, i.e., it is not initialized. Furthermore, we define the operator $\circ$ to combine variables that may have the value $\perp$ as follows: $\perp \circ \perp = \perp$ and $a \circ \perp = \perp \circ a = a \circ a = a$ for any value $a \neq \perp$. Combining two different values $a \neq b$ with $a, b \neq \perp$ is undefined. We also use the element-wise OR operator $\oplus$, which operates on binary values with $0 \oplus 0 = 0$ and $1 \oplus 0 = 0 \oplus 1 = 1 \oplus 1 = 1$.

Assumptions. We assume that any communication in any phase of the protocol can fail. This also includes the possibility that a node can neither receive from nor send any packet to other nodes. We assume a non-Byzantine failure model, i.e., a packet may either not be received at all or received correctly. Technically, a corrupted packet can be classified as not received through the use of cyclic redundancy checks (CRCs).

1.1 Terminology

We define the following terms which will be used both for the protocol explanation and the proofs as follows:

- A *strongly connected component* (SCC) is a subset of nodes $S \subseteq V$ of a directed graph $G = (V, E)$ where a path exists between each pair of vertices $v, w \in S$. We will also refer to this as a *sub-network*.
- A *majority strongly connected component* (MSCC) is an strongly connected component (SCC) with $|S| > N/2$, i.e., an SCC which contains a majority of nodes.
- A *network* is an MSCC that does not have any edges leaving the component, i.e., $\nexists\, v \in S$ such that $\exists\, (v, w) \in E$ with $w \notin S$. This is equivalent to the requirement that the MSCC is a leaf node of the condensation of G, which we will also call *maximal* SCC.
- A *complete node* is a node which has obtained the complete set of information, i.e., for which $\forall j \in [1, N]$ we find $M'[i] = 1 \Rightarrow R'[j] \neq \perp$. This is equivalent to the statement that the node obtained requests from all nodes inside its network.

1.2 Protocol requirements

We are interested in three major protocol requirements related to fault tolerance:

Safety We require that no two nodes send data in the same DD slot, i.e., no packet collisions with other Hydra nodes occur and therefore freedom from network-internal interference is ensured. Moreover, if two nodes distribute a schedule in the SD phase, they distribute the same one, i.e., freedom from scheduling conflicts.

Adaptivity The schedule of the DD phase can dynamically adapt to changing demands and environmental conditions even if there are intermittent or persistent communication errors.

Liveness Synchronized nodes continue to exchange data in the DD phase as long as communication is physically possible, even if a new schedule could not be obtained or if there are intermittent or persistent communication errors.

These three requirements are in conflict with each other. As a design choice, we unconditionally guarantee the safety of Hydra during its entire operation including bootstrapping. Preventing packets from colliding with transmissions of other nodes in the network is a precondition for highly reliable packet delivery. If we were to relax this requirement, then achieving liveness and adaptivity would be easily possible using best-effort strategies, e.g., letting nodes just send data packets according to their communication demand.

In addition, there is a trade-off between liveness and adaptivity. If we permanently fix a schedule, then safety and liveness can be guaranteed as the schedule is static and known to every node. On the other hand, when we dynamically adapt the schedule according to the demands of nodes, then either safety or liveness is threatened. To fulfill all requirements simultaneously, we require consensus among nodes on a new schedule, i.e., which requests are considered. Otherwise, safety may be violated in the SD phase if different schedules are distributed. And even if such consensus and unique scheduling are guaranteed, distributing a schedule may still compromise safety in the DD phase if we do not know with certainty that all available nodes have received it, since nodes with an old schedule still participate in data communication due to liveness. Hydra includes parameters that allow adjusting the trade-off between liveness and adaptivity while continuously guaranteeing safety.

2 Bootstrapping Mechanism of Hydra

For nodes to form a network, Hydra requires a mechanism to start a network from scratch. The bootstrapping algorithm, presented in Algorithm 1, is opportunistic and does not fulfill the requirements as presented in Section 1.2. However, as it can be arbitrarily restarted without affecting the performance of Hydra's normal operation, it guarantees synchronization after termination and enables liveness.

A prerequisite for Hydra's algorithm presented in Algorithm 2 and Algorithm 3 is that all nodes are synchronized, i.e., that they know the starting times of rounds as well as the epoch offset f. This assumption is not yet fulfilled while the network is starting up. Furthermore, when including clock drift, there are other situations where established synchronization can be lost, namely when there is no functioning sub-network with more than $N/2$ nodes left, or when there are two or more separate sub-networks that do not hear each other and whose clocks drift apart.

2.1 Concepts

We propose the following bootstrapping mechanism to overcome this difficulty:

- We introduce the state *synced* which is initially set to $synced \leftarrow 0$. The variable is set to $synced \leftarrow 1$ if a node receives a packet from an existing network in the YD phase in line 7. The variable is set to $synced \leftarrow 0$ if a node is not part of an operating network for a certain number of rounds, i.e., if line 8 in Algorithm 3 is executed after the check in line 5 detected no interaction with a majority of nodes for E_{max} rounds in a row. The threshold E_{max} depends on the maximal clock drift as well as the round period T.

- If there is no existing network, then no packets will be distributed on the main channel to which nodes could synchronize. As the process of synchronization may lead to interference

Algorithm 1 Bootstrapping mechanism

YD (synchronization distribution) phase
  $c \leftarrow PICK(\{c_{main}, c_{boot}\})$; $t_{listen} \leftarrow PICK([0, K)) \cdot T_{slot}$;
  ▷ Check if node can synchronize to others ◁
  if (packet q received within t_{listen}) **then**
    synchronize to packet ;
    if $(c = c_{main})$ **then**
      ▷ Initialize parameters for Algorithm 2 ◁
      $v \leftarrow 0$; $f \leftarrow EPOCH\_OFFSET(q)$;
      synced $\leftarrow 1$; *updated* $\leftarrow 0$; *unchanged* $\leftarrow 0$; *retransmit* $\leftarrow 0$;
      for all $k \in [1, K]$ **do**
        $S^v[k] \leftarrow \perp$;
      for all $j \in [1, N]$ **do**
        $C_e[j] \leftarrow 0$; $I_e[j] \leftarrow 0$; $M[j] \leftarrow 0$;
      switch to Algorithm 2 ;
  else if $(c = c_{main})$ **then**
    restart Algorithm 1 ;
  else
    send sync packet ;

YN (synchronization negotiation) phase
  Initialization
    synced $\leftarrow 0$;
    for all $j \in [1, N]$ **do**
      $M[j] \leftarrow \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$;
  Sending
    send packet $p = (M)$;
  Receiving
    if (packet $q = (M_q)$ received) **then**
      $M \leftarrow M \vee M_q$;
  Final
    ▷ Check if majority was found to start new network ◁
    if $(\sum_{j \in [1,N]} M[j] > N/2)$ **then**
      synced $\leftarrow 1$;

YD (synchronization distribution) phase
  if $(synced = 1)$ **then**
    send sync packet p ; $\tau \leftarrow TX\_TIME(p)$;
  else if (sync packet q received) **then**
    synced $\leftarrow 1$; $\tau \leftarrow TX\_TIME(q)$;

End of round
  ▷ Check if bootstrapping has been successful ◁
  if $(synced = 1)$ **then**
    ▷ Initialize parameters for Algorithm 2 ◁
    wait until $\tau + \Delta\tau$;
    $v \leftarrow 1$; $f \leftarrow 0$; *updated* $\leftarrow 0$; *unchanged* $\leftarrow 0$; *retransmit* $\leftarrow 0$;
    for all $k \in [1, K]$ **do**
      $S^v[k] \leftarrow \perp$;
    for all $j \in [1, N]$ **do**
      $C_e[j] \leftarrow 0$; $I_e[j] \leftarrow 0$;
    switch to Algorithm 2 ;

due to its asynchronous communication, we use a second frequency channel, denoted as *boot channel*. Nodes with *synced* = 0 not only listen on the main channel during the `YD` phase but can also try to synchronize with other nodes on the boot channel. To discover other nodes that are bootstrapping, they start using the boot channel with increasing probability and at the same time decrease the time spent listening on the main frequency channel. If they are able to synchronize with more than $N/2$ other nodes, they set *synced* $\leftarrow 1$ in line 29 and start a new network at the end of the round in line 43. While this results in potentially more than $N/2$ nodes participating in the `SN` phase, there is no risk of interference on the main frequency channel since no existing network can continue operating as they lack a majority of nodes.

- The bootstrapping algorithm is deliberately designed to be highly similar to the `SN` and `SD` phases in Algorithm 2 to minimize engineering overhead, but now serves the purpose of synchronization. We call the phases `YN` (synchronization negotiation) and `YD` (synchronization distribution). A node with *synced* = 0 participates in boot channel communication with increasing probability. The details of the phases are presented in Algorithm 1.

2.2 Protocol description & explanation

When starting the bootstrapping mechanism, each node picks a channel $c \in \{c_{main}, c_{boot}\}$ as well as a time $t_{listen} \in [0, K) \cdot T_{slot}$ at random in line 2. Thereafter, it listens for t_{listen} on channel c. If a node receives a message on the main channel, it knows that an already synchronized network exists and switches to Algorithm 2 in line 12. To acquire the necessary state for this transition, the function $EPOCH\_OFFSET$ utilizes the timestamp contained in any Hydra packet and modulo computation to derive the current round r and epoch offset f in line 6. A node that did not hear anything can initiate the flood of a first sync packet during the `YD` phase where it announces its presence in line 16. This broadcast can then be used by receiving nodes to synchronize for the following `YN` and `YD` phases in line 4.

The node that initiates the flood during the `YD` phase as well as the nodes that have received it participate in the following `YN` phase by only setting their own membership flag (line 21) and exchanging information just as during an `SN` phase. M collects the information about participating nodes in line 26. The purpose of this phase is to let nodes know whether there are at least $N/2$ nodes participating in the current boot round. If this can be asserted, the corresponding nodes are in sync (line 29) and send a second sync packet in the final `YD` phase in line 32. Receiving nodes set *synced* $\leftarrow 1$ in line 34. They determine the local time τ when the sync packet was initiated using the function TX_TIME and start a communication round at local time $\tau + \Delta\tau$, where $\Delta\tau \geq F \cdot T$ is a predefined constant and guarantees that any potential previous network has forced its schedule to expire before a new network can start operating. Nodes that still have *synced* = 0 may initiate another boot round at some random time later.

3 Extended Algorithm of Hydra

The algorithm underlying Hydra has already been presented in our conference publication. In this section, we first present an extended version of the algorithm with additional functions delivering more configuration options and improving its energy efficiency. Thereafter, we highlight some of the technical details for the implementation of Hydra in Section 3.3.

3.1 Concepts

An important concept that enables liveness and adaptivity despite node failures is the following: Nodes in the network are using at most two different schedule versions at the same time, and these schedules are compatible and lead to a safe `DD` phase. To enable this, we must preserve

state both across multiple protocol phases as well as rounds. Each node of the network has the following state information that is maintained across rounds:

- Each node knows its static ID $1 \leq i \leq N$, the number of nodes N, the round period T, and the length of an *epoch* $F \in \mathbb{Z}_{>0}$.
- We define the *schedule version* $v \in \mathbb{Z}_{\geq 0}$ that counts the computed schedules. Initially, we set $v \leftarrow 0$, i.e., the schedule is not valid.
- Each node stores its current *schedule* S^v, where $S^v[k] \in \{\perp\} \cup [1, N]$ denotes the allocation of `DD` slots $k \in [1, K]$ to nodes. Initially, we set $S^1[k] \leftarrow \perp$ for all slots k in the `DD` phase, i.e., slots are not assigned to any node.
- Before starting a new round, each node knows its own *request* r_i^v for the next schedule. It must be unique for every schedule version v, i.e., it can only be adjusted when the schedule is updated. This request can be formulated in any data structure and contain e.g. release time, deadline, packet size, and period.
- We use the three threshold values $E_{max} \in \mathbb{Z}_{>0}$, $C_{join} \in \mathbb{Z}_{>0}$, and $C_{stay} \in \mathbb{Z}_{>0}$ to configure the duration during which nodes can remain within a network until expiration and the requirements for nodes to join and stay within a network.

In addition to the long-term state information, the algorithm also uses some intermediate variables during a round which may be kept up to the end of an epoch:

- We define the current *epoch offset* $f \in [0, F]$, which is initially set to $f \leftarrow 0$.
- We introduce the temporary *schedule requests* R' with elements $R'[j]$ for $j \in [1, N]$. $R'[j] = \perp$ denotes that no request information is available from node j and otherwise, $R'[j]$ is the request r_j^v of node j. We initially set $R'[j] \leftarrow \perp$ for all $j \in [1, N]$.
- We introduce the *membership flags* M with $M[j] \in \{0, 1\}$ for all $j \in [1, N]$. Initially, we assume $M[j] \leftarrow 0$ for all $j \in [1, N] \,|\, j \neq i$ and gather information on connected nodes both during Algorithm 1 and Algorithm 2. During the `SN` phase, we also use the *temporary membership flags* M' with equivalent formalism.
- We introduce two *epoch counters* C_e and I_e with $C_e[j] \in [0, F]$ and $I_e[j] \in \{0, 1\}$ for all $j \in [1, N]$. In addition, we also introduce a corresponding *round counter* C_r with $C_r[j] \in \{0, 1\}$ for all $j \in [1, N]$. Initially, we set $C_e[j] \leftarrow 0$, $I_e[j] \leftarrow 0$, and $C_r[j] \leftarrow 0$ for all $j \in [1, N]$.

Finally, the following three additional binary state variables are used:

- The state variable *updated* $\in \{0, 1\}$ denotes whether a new schedule could be determined. It is initially set to *updated* $\leftarrow 0$.
- The state variable *unchanged* $\in \{0, 1\}$ denotes whether a newly computed schedule corresponds to the previous one and hence must not be distributed, with the schedule version remaining the same. It is initially set to *unchanged* $\leftarrow 0$.
- The state variable *retransmit* $\in \{0, 1\}$ denotes whether a schedule must be re-transmitted as some nodes have not received it during past rounds and must be able to catch up. It is initially set to *retransmit* $\leftarrow 0$.

We now define the primary algorithm behind Hydra in Algorithm 2 and Algorithm 3 (split up for convenience) and then explain some of its elements.

Algorithm 2 Protocol behavior of node i after bootstrapping

DD (data dissemination) phase
  Every node i with $v > 0$ participates with its current schedule S^v ;

SN (schedule negotiation) phase
  Initialization
    $v^{min} \leftarrow v$; $v^{max} \leftarrow v$;
    for all $j \in [1, N]$ **do**
      $M'[j] \leftarrow M[j]$; $R'[j] \leftarrow \begin{cases} r_i^v & \text{if } j = i \\ \perp & \text{otherwise} \end{cases}$;
      $C_r[j] \leftarrow 0$;
  Sending
    send packet $p = (i, v^{min}, v^{max}, M', R')$;
  Receiving
    if (packet $q = (j_q, v_q^{min}, v_q^{max}, M_q, R_q)$ received) **then**
      for all $j \in [1, N] : R_q[j] \neq \perp$ **do**
        $C_r[j] \leftarrow 1$;
      if $((M'[j_q] = 1) \wedge (M_q[i] = 1))$ **then**
        for all $j \in [1, N] : R_q[j] \neq \perp$ **do**
          $I_e[j] \leftarrow 1$;
        $v^{min} \leftarrow \min\{v^{min}, v_q^{min}\}$; $v^{max} \leftarrow \max\{v^{max}, v_q^{max}\}$;
        $R' \leftarrow R' \circ R_q$; $M' \leftarrow M' \vee M_q$;
  Final
    if $((\forall j \in [1, N] : M'[j] = 1 \Rightarrow R'[j] \neq \perp) \wedge$
       $(|\{j \in [1, N] : M'[j] = 1\}| > N/2))$ **then**
      if $(v^{min} = v^{max})$ **then**
        if $(v > 0)$ **then**
          determine new schedule S ;
          ▷ Check if schedule remained the same ◁
          if $(S^v = S)$ **then**
            unchanged $\leftarrow 1$;
          else
            updated $\leftarrow 1$;
        else
          ▷ All nodes inside the network had to expire their schedule ◁
          synced $\leftarrow 0$; switch to Algorithm 1 ;
      else if $(v = v^{max})$ **then**
        retransmit $\leftarrow 1$;
    for all $j \in [1, N]$ **do**
      $C_e[j] \leftarrow C_e[j] + C_r[j]$;

SD (schedule distribution) phase
  if $((f = F - 1) \wedge (\textit{updated} = 1))$ **then**
    $v \leftarrow v + 1$; $S^v \leftarrow S$; send packet (v, S^v) ;
  else if $(\textit{retransmit} = 1)$ **then**
    send packet (v, S^v) ; *retransmit* $\leftarrow 0$;
  ▷ Check if schedule does not change ◁
  else if $(\textit{unchanged} = 1)$ **then**
    wait until the end of the phase ;
  else if (packet $q = (v_q, S_q)$ received) **then**
    $v \leftarrow v_q$; $S^v \leftarrow S_q$;
    for all $j \in [1, N]$ **do**
      $I_e[j] \leftarrow 1$;

Algorithm 3 Protocol behavior of node i at the end of the round

1: **End of round**
2: $f \leftarrow f + 1$;
3: **if** $(f = F)$ **then**
4: $updated \leftarrow 0$; $unchanged \leftarrow 0$; $f \leftarrow 0$;
5: **if** $(\sum_{j\in[1,N]} I_e[j] \leq N/2)$ **then**
6: $v \leftarrow 0$; $E \leftarrow E + 1$;
▷ Check if node has lost connection to the network ◁
7: **if** $(E \geq E_{max})$ **then**
8: $synced \leftarrow 0$; switch to Algorithm 1 ;
9: **else**
10: $E \leftarrow 0$;
11: **for all** $j \in [1, N]$ **do**
▷ Check if nodes newly joined or stayed ◁
12: **if** $(((M[j] = 0) \wedge (C_e[j] \geq C_{join})) \vee$
$((M[j] = 1) \wedge (C_e[j] \geq C_{stay})) \vee$
$(j = i))$ **then**
13: $M[j] \leftarrow 1$;
14: **else**
15: $M[j] \leftarrow 0$;
16: $C_e[j] \leftarrow 0$; $I_e[j] \leftarrow 0$;

3.2 Protocol description & explanation

Phases. During the DD phase, all nodes which know a valid schedule, i.e., for which $v > 0$, can actively participate and initiate floods. If a node does not have a valid schedule, it might still assist in flooding packets from other nodes or might simply passively listen to transmissions. As nodes always participate in the DD phase (line 2), liveness is ensured. Safety of the DD phase requires that the schedule versions that are present in the nodes are compatible, i.e., they do not lead to slots that are assigned to two different nodes. This is achieved by (i) guaranteeing that at most two successive schedules are present, and (ii) that successive schedules are compatible.

The SN phase initializes the bounds on the schedule versions v^{min} and v^{max} in line 5 with the current local schedule version v. These version bounds v^{min} and v^{max} denote the minimal and maximal versions seen in this round. In line 7, the schedule requests initially just contain the request of the node itself, namely $R'[i] = r_i^v$. The following sending and receiving actions are executed concurrently. A node continuously sends its version bounds and schedule requests in line 10 and updates its local bounds and schedule requests in lines 18-19 upon receiving information from another node. Updates in lines 17-19 only happen if node i is part of the current membership flags of j and vice versa, i.e., if nodes i and j "know each other" and consider each other as being part of the same network. In line 19, the membership flags M' are updated in addition to the current schedule requests R', i.e., if nodes i and j are part of the same network and i receives information from j, then i includes node j's view of the network in its own by complementing M'. As a result, it fills the schedule request variable R' with requests from other nodes and its version bounds v^{min} and v^{max} contain the minimal and maximal schedule version from all nodes j with $R'[j] \neq \perp$. If all nodes have the same schedule version (line 22), a new schedule S is computed in line 24 by all the nodes that have obtained the complete set of information, which is defined as $\forall j \in [1, N] : M'[i] = 1 \Rightarrow R'[j] \neq \perp$. The state $updated \leftarrow 1$ stores this event in case the schedule differs from the previously determined one. Otherwise, $unchanged \leftarrow 1$ informs the node that it does not have to spend more energy during the SD phase as no changes must be distributed or will be received. If the complete set of information could be obtained but the existence of an old schedule inside the network is detected, $retransmit \leftarrow 1$ in line 32 will trigger a re-transmission if the node is in possession of the most up-to-date schedule and permits other nodes to catch up. We conclude that a new

schedule is only determined in line 24 if three conditions are satisfied: (*i*) The node received the complete set of information from its own sub-network, i.e., it received the requests from all nodes of the network, (*ii*) the network contains more than $N/2$ nodes (i.e., it is a majority network), and (*iii*) all nodes in the network have the same schedule version. Condition (*i*) guarantees that all nodes that compute a new schedule do so based on the same set of requests and therefore, the new schedule is identical. Condition (*ii*) guarantees that there are never multiple sub-networks that independently compute new schedules. Condition (*iii*) guarantees that nodes computing the new schedule do so based on the same previous schedule.

The `SD` phase primarily serves as a means to also distribute the schedule to nodes that do not have the newest version, either because they newly joined or as they missed some packets in the past. If all nodes in the network have the same schedule version, nodes with the complete set of information send their updated schedule (line 37) at the end of an epoch (line 36) as previously calculated in line 24. If a node recognizes that it has a more recent schedule version than some of the nodes from which it has received schedule requests, it re-transmits its schedule version and schedule in line 39. Several nodes may satisfy this condition and there is a potential safety violation if their schedule versions do not match. However, as will be shown later, there exist at most two schedule versions at any point in time in the network. As a result of the re-transmission, all nodes may have the most recent schedule in the next round. If nodes do not satisfy any of the conditions for sending a schedule and have not asserted that no changes need to be made, they may receive a broadcasted schedule in line 43.

Schedule compatibility. Informally, we now show that (*i*) there are at most two schedule versions in the network and (*ii*) how we can compute a new schedule version in line 24 such that two successive schedules are compatible and do not lead to safety violations in the `DD` phase.

Let us suppose that at the beginning of a round, there are two schedule versions v and $v-1$ known to nodes of the network. Then, a new schedule version cannot be computed in line 24 due to the check in line 22, and therefore, no updated version can be distributed in line 37. The computation of a new schedule can only occur after all nodes with $v-1$ received version v in line 43, after which version $v+1$ can be computed. In both cases, there are at most two schedule versions in the network, and they are successive.

In order to ensure compatibility of successive schedules S^v and S^{v+1}, certain conditions must be satisfied. Otherwise, nodes with successive schedules could send data in the same slot, resulting in a safety violation during the `DD` phase. First, note that, in terms of the schedule requests, the order of slots k is secondary and only the number of slots allocated to a node j is relevant. Suppose that we are given some schedule S^v that was based on previous requests, and suppose that the new schedule requests R' lead to a different number of slots allocated to node j. To update the schedule, we free all slots of S^v that are not required anymore to reach the number of slots allocated to each node. For example, if the number of slots allocated to node j decreases by δ_j, then δ_j slots of j are released. If another node k receives δ_k more slots in the new schedule, we assign slots to node k that are free in S^v. This means that a slot that is assigned in S^v can only be used by the same node in S^{v+1} or be released; on the other hand, a free slot in S^v can be assigned to any node in S^{v+1} or remain free. To be more formal, to ensure compatibility of two successive schedules S^v and S^{v+1}, we require for $i \neq \perp$:

$$S^v[k] = i \implies S^{v+1}[k] \in \{i, \perp\}$$

As a result, we can transition from any slot assignment to any other assignment in no more than two consecutive schedules (in the worst case using the empty schedule as an intermediate step), thereby providing adaptivity while ensuring safety in the `DD` phase and permitting an asynchronous schedule update at any node.

Epochs. To provide further stability and avoid rash decisions which might jeopardize performance, Hydra uses epochs to aggregate information over multiple rounds before adapting. Each epoch consists of F rounds, and a new schedule is only distributed in the last round of an epoch,

as verified in line 36. During an epoch, the protocol collects information on (i) which nodes will be part of the current network and (ii) which nodes will be allowed to send during the DD phase, i.e., only nodes with a schedule version $v > 0$ (line 2).

The SN phase initializes the temporary membership flags M' to the current membership flags M (line 7) in each round and updates them during the SN phase in line 19. Intuitively, $M[j] = 1$ denotes that node j is part of the perceived network of node i. However, we require a mechanism to update the membership flags. For this, the variable C_r is reset at the beginning of the phase. It collects connectivity information during the SN phase and reflects whether node i receives information from node j. The variable C_e then collects information from C_r throughout the epoch. In line 34, $C_e[j]$ is increased by 1 if i received information from j. This information will later be used to decide on updating the current sub-network of node i, i.e., the membership flags M. Similarly, the variable I_e contains interaction information and reflects whether node i integrated and merged information from node j. Just as C_e, I_e is an epoch counter and only resets at the end of the epoch. Its content will influence whether a node can continue to participate in the DD phase or whether its schedule has to expire in line 6.

During the SD phase, the new schedule is only distributed in the last round of an epoch in line 37. This mechanism avoids interference during the SD phase with nodes that still use an outdated schedule version until it expires (see below). If a node receives a new schedule within a round, it knows that it has obtained the most recent schedule which is in use. Therefore, this schedule is guaranteed to be valid until the end of the next epoch and we set $I_e[j] \leftarrow 1$ for all $j \in [1, N]$ in line 45 to avoid the expiration of the schedule version in line 6.

At the end of an epoch ($f = F$), we reset various variables such as the epoch offset f and the binary states *updated* and *unchanged* in line 4, as well as C_e and I_e in line 16. If node i cannot assert that it is still part of a network containing the majority of nodes, it sets $v \leftarrow 0$ in line 6 as it may be disconnected and have an outdated schedule version. $v = 0$ guarantees safety in the forthcoming DD and SD phases. In line 13, we set the new membership flag $M[j] = 1$, i.e., node i is connected to node j, if one of two conditions are satisfied: If node i was not connected to node j beforehand ($M[j] = 0$) but received information from this node at least C_{join} times during the current epoch, the connection is added and a new node joins the network. If node i was previously connected to node j ($M[j] = 1$) and received information from this node at least C_{stay} times, the connection is maintained and j stays in the network. Therefore, C_e serves as a means to update the current network structure and offers flexibility to the system designer. Usually, one might set $C_{stay} \leq C_{join}$ so a node that already joined the network is likely to stay and scheduling is not disturbed. This way, decoupling these two thresholds permits a high entry level to the network (by increasing C_{join}) while preserving network stability.

3.3 Technical details

Packet structure. As the data exchange in the DD phase consists of normal Glossy floods, Hydra requires no adjustments to the data packets. The packets in the SN phase contain $N/8$ B for the membership flags and N requests of 3 bits each to add or remove up to 3 slots per node and schedule version. Including 2 B for the extremal values of the schedule version, 14 B are sufficient to obtain the complete set of information of a network with $N = 24$ nodes. However, we encountered occasional bit flips which were not caught by the standard 16-bit hardware CRC appended by the radio. As the distributed consensus mechanism relies on the correctness of the packet content and does not consider Byzantine failures, we included an additional 32-bit CRC in the payload that successfully eliminates corrupted packets. Lastly, the schedule in the SD phase is compressed to $log(N)$ bits per DD slot and fits into 50 B for 80 slots and $N = 24$.

SN propagation policy. As Hydra does not rely on the successful completion of the SN phase for liveness and safety, the propagation policy according to which a node either transmits or receives in a Chaos slot is flexible. We opt to remain close to the standard Chaos policy [Landsiedel

et al., 2013] to achieve reliable performance but adapt the initiation to avoid a single point of failure (SPOF). Through randomized TX/RX decisions with a fixed TX probability of 25% until the first successful reception, Hydra begins without relying on a dedicated node. This concurrent start is only made possible by the tight time synchronization obtained through the preceding Glossy floods. Thereafter, we rely on more selective transmissions either after a randomized timeout of 3 to 5 slots or after a packet with new information has been received so that its content can quickly propagate to neighbors which are likely also lacking this information. Nodes that obtain the complete set of information aggressively transmit 5 consecutive times and thereafter fall back to randomized TX/RX decisions if they detect that others are still missing information. Policies considering the local node density, such as the one presented by Mixer [Herrmann et al., 2018], could further improve performance at the cost of delayed adaptivity as they require gathering reliable neighborhood information over multiple rounds.

Finite schedule versions. Our packet structure uses 1 B to store a schedule version which should monotonically increase with each new schedule computation but is limited by the variable size. To solve this issue, we leverage the fact that we know that the difference between schedule versions in a network is bounded to two. This allows us to apply modulo arithmetic and preserves the correctness of comparisons if the version exceeds the variable size.

Time synchronization. Without a unique time reference whose long-term availability is guaranteed, clock drift compensation becomes challenging as a node keeps synchronizing with a varying set of nodes. Nevertheless, continuous clock correction is important to maintain reliable communication even if synchronization is temporarily prevented. Therefore, we collect the clock tick shifts that a node applies per round when it re-synchronizes to others and compute a local correction factor. However, it may be that groups of nodes observe a perpetually increasing or decreasing clock drift. To maintain a stable clock speed, we exchange the local correction factors in the `SN` phase and aggregate the extremal values. This network information is used to estimate an average correction factor as the mean of the minimum and maximum and normalize the local factor that nodes apply to stabilize the clock drift.

As an additional note, it should be remarked that synchronization for the `SN` phase can be improved if the last few `DD` slots are assigned and used. This follows as these slots are used by all nodes in the network to re-synchronize and provide a highly reliable and precise synchronization for the contention-based communication following directly thereafter. Therefore, our implementation of the scheduling algorithm prefers to assign slots toward the end of the round and fills the schedule starting at the end. Note that this does not restrict scheduling as it is entirely optional and does not influence any other features of Hydra, but that it can be used as an orthogonal consideration when designing a scheduling algorithm for Hydra.

Direct slot transfer. In order to permit a transfer of schedule slot assignments within a single schedule update, we can optionally introduce a scheduling policy where nodes can entirely dictate the number of released slots through their requests. This means that if a node's request $\delta < 0$, the node will already stop using the first δ slots before it transmits the request. During the schedule computation, we can then first free all released slots and thereafter directly reallocate these (already unused) slots. As a node knows that it can dictate the number of released slots and already stops using them before the new schedule is computed, successive schedules still do not cause interference even if the releasing node might not be able to update its schedule and will continue using the previous version. Notice that this does not require any assumptions on whether additional slots are granted and is purely based on a node releasing its own slots earlier than required by the schedule.

4 Proofs

For some of the following proofs, we extend the notation of variables by either (i) adding the index of a node to the relevant quantities in subscript, e.g. R'_i denotes the schedule requests at node i, or (ii) adding the schedule version in superscript, e.g. M^v denotes the membership flags which were used to generate schedule version v.

4.1 Safety

For Hydra's distributed consensus mechanism, we rely on the fact that for arbitrary initial membership flags M_k stored in a node k (lines 13, 15), i.e., independent of the set of nodes that are assumed to be reachable by node k, the schedule requests used to compute the next schedule (line 24) at node k, i.e., $R'_k[j]$ for all j, are the same for all nodes k with a complete set of information. In short, all nodes with a complete set of information compute the identical schedule.

Theorem 4.1 (Consensus on current network)**.** *After the* `SN` *phase and independently of the initial membership flags M_k of any node $k \in [1, N]$, any two nodes $i, j \in [1, N]$ with a complete set of information either satisfy $R'_i = R'_j$ (identical requests) or $R'_i[k] \neq \perp \Rightarrow R'_j[k] = \perp$ for all $k \in [1, N]$ (requests from disjoint sets of nodes).*

Proof. To prove the statement, we use a graph interpretation of the protocol where each node corresponds to a node $i \in V$ in a directed graph $G = (V, E)$. The initial membership flags M_i lead to the edges E of G: If $M_i[j] = 1$, then $(i, j) \in E$. Note that $(i, i) \in E$. We associate two sets with each node $i \in V$:

- $m_i \subseteq V$, where initially we have $m_i := \{j : (i, j) \in E\}$. This set corresponds to M'_i, i.e., we have $j \in m_i \Leftrightarrow M'_i[j] = 1$.
- $r_i \subseteq V$, where initially we have $r_i := \{i\}$. This set corresponds to R'_i, i.e., we have $j \in r_i \Leftrightarrow R'_i[j] \neq \perp$.

We perform arbitrary updates on these sets according to the protocol. In particular, node i may merge information with a node j if $j \in m_i \wedge i \in m_j$ (line 15). If so, the sets of node i are updated to $m_i \leftarrow m_i \cup m_j$ and $r_i \leftarrow r_i \cup r_j$. A node has a complete set of information if $m_i = r_i$. Using this graph representation, we now prove an equivalent statement:

> *Independently of G and the number and order of merges, any two complete nodes $i, j \in V$ either satisfy $r_i = r_j$ or $r_i \cap r_j = \emptyset$.*

To show this result, we apply three invariants:

1. If $k \in r_i$, then $l \in m_i$ for all l with $(k, l) \in E$, i.e., if a node k is in the set r_i, then not only is this node in set m_i but also all its immediate successor nodes. This also leads to $r_i \subseteq m_i$. The proof of this property can be done using induction with the initial sets m_i and r_i as a base case. Supposing that the property holds for any nodes i and j, and, without loss of generality, that information of node j is joined with the information at node i. Then we can show that it holds for the new sets of node i after joining, namely $m'_i := m_i \cup m_j$ and $r'_i := r_i \cup r_j$. If $k \in r'_i$, then either $k \in r_i$ or $k \in r_j$. If $k \in r_i$, then all immediate successors are in m_i and therefore also in m'_i. If $k \in r_j$, then all immediate successors are in m_j and therefore also in m'_i.
2. If $k \in m_i$, then there exists a directed path in G from i to k. Initially, this property holds as $k \in m_i \Leftrightarrow (i, k) \in E$. Supposing that the property holds for all m_j and that j joins its information with i, we then show that the property also holds for the new

set $m_i' := m_i \cup m_j$. If $k \in m_i'$, then either $k \in m_i$ or $k \in m_j$. In the first case, the property holds due to the assumption. In the second case, note that $j \in m_i$ due to the join condition and therefore, there exists a path $i \longrightarrow j$ and a path $j \longrightarrow k$ which leads to a path $i \longrightarrow k$ after joining.

3. If $k, l \in r_i$, then there exists a directed path in G from k to l. Supposing that the property holds for all r_j and that j joins its information with i, we then show that the property also holds for the new set $r_i' := r_i \cup r_j$. If $k, l \in r_i$ or $k, l \in r_j$, then the property holds due to the assumption. Now, without loss of generality, consider that $k \in r_i$ and $l \in r_j$. Note that due to the join condition, we have $i \in m_j$ and $j \in m_i$ and therefore, we know from the statement above that paths $i \longrightarrow j$ and $j \longrightarrow i$ exist. Moreover, $i, k \in r_i$ and $j, l \in r_j$. Therefore, we can conclude that there are paths $k \longrightarrow i \longrightarrow j \longrightarrow l$ and $l \longrightarrow j \longrightarrow i \longrightarrow k$, which proves the property.

Based on the above properties, we can conclude that if a node has the complete set of information ($m_i = r_i$), the nodes in m_i are strongly connected (each node $k \in m_i$ can reach any other node $l \in m_i$), and for any node in m_i, all its immediate successors are also in m_i. As a result, m_i of a complete node i forms an SCC of G. As all immediate successor nodes are already in m_i and therefore no node can be added, it is a *maximal* SCC. As such SCCs form a partition of all nodes [Tarjan, 1972], each node $i \in V$ can be in exactly one SCC, which proves the property. □

Corollary 4.1 (Uniqueness of schedule computation)**.** *Any two nodes $i, j \in [1, N]$ which have obtained the necessary information to compute a new schedule will compute the same schedule with the same schedule version.*

Proof. We know from Theorem 4.1 that such nodes must have complete sets of information that are either identical or disjunct. Because only a single disjunct set can contain a majority of nodes required for schedule computation (line 21), at most one of them can compute a new schedule if their requests are disjunct. Only nodes that share the complete set of information of the MSCC as defined in Theorem 4.1 have the preconditions for schedule computation, with all such nodes in the current network having the same set of schedule requests R' and the same schedule version (line 22). As the scheduling algorithm is deterministic, they compute the same schedule S with the new schedule version $v' := v + 1$, which guarantees the uniqueness of a newly computed schedule. □

Based on the above result, the following two theorems show the safety of Hydra, i.e., no two nodes send different packets at the same time in the `SD` phase (Theorem 4.2) or `DD` phase (Theorem 4.3).

Theorem 4.2 (Safety of schedule distribution)**.** *In the `SD` phase, no two nodes $i, j \in [1, N]$ transmit packets with different contents (v, S^v) (lines 37 and 39 of Algorithm 2).*

Proof. Only nodes that have received the identical complete set of information satisfy either *updated* $= 1$ or *retransmit* $= 1$ (see proof of Corollary 4.1) and send packets in the `SD` phase. If the condition in line 22 is satisfied, we know from Corollary 4.1 that the resulting schedule is unique and is distributed in line 37. If the condition in line 31 is satisfied, there exists exactly one maximum schedule version v_{max} for the set of nodes in the current network which obtained the complete set of information. This follows as the element-wise max operator is idempotent and commutative, and therefore all nodes in the MSCC which obtain the complete set of information compute the same v_{max}. From these nodes, only the subset which knows the corresponding schedule will transmit it. Notice that as reaching line 31 is mutually exclusive with fulfilling the condition in line 22 and as all nodes in the network operate on the same set of information, only one of these two cases can occur within the network, which proves the statement. □

Theorem 4.3 (Safety of data dissemination)**.** *In the* DD *phase, no two nodes* $i, j \in [1, N]$ *transmit different packets in the same* DD *slot (line 2 of Algorithm 2).*

Proof. According to the scheduling conditions, two successive schedule versions can be used by any nodes $i, j \in [1, N]$ without violating safety. Note that nodes with $v = 0$ do not participate in the DD phase and are safe (line 2). Therefore, we need to show that at the beginning of the DD phase, nodes have either successive schedule versions or satisfy $v = 0$. We will prove this theorem by induction, where the schedule with version $v^* = 2$, the first one involving requests from other nodes, as well as the empty schedule with version $v^* - 1 = 1$ originating from the bootstrapping protocol (line 38) serve as a base case.

Suppose that at the start of a round, two schedule versions v^* and $v^* - 1$ are in use. Let us first look at nodes within the current network, i.e., the SCC of G in the proof of Theorem 4.1. Due to the mismatch in used schedule versions, the next schedule cannot be computed in line 24 and hence cannot be distributed in line 37. Only after all nodes with $v^* - 1$ have received version v^* in line 43 in a future round (and $v^* - 1$ is not used anymore), a new schedule $v^* + 1$ can be computed as an induction step. In both cases, there are at most two schedules used by nodes within the current network, and they are successive.

A node that is not part of the current network may still know the outdated schedule version $v^* - 1$ if it did not receive any of the newer schedules distributed in the SD phases. The continued use of $v^* - 1$ is prevented by forcing $v \leftarrow 0$ in line 6 as neither has a schedule been received in any SD phase of the current epoch nor is the node part of a majority network, i.e., an SCC of G with more than $N/2$ nodes. The latter is derived in line 5 from variables $I_e[j]$ which are only set in line 17 if the node has a bidirectional path to a node j in G. As the node is not part of the current network anymore, it can only be part of a sub-network containing a minority of nodes and hence cannot acquire sufficient interactions to prevent such a schedule expiration. □

As a result, nodes can independently and asynchronously update their schedules and remain assured that their transmissions will not collide with data packets from nodes using another schedule.

Theorem 4.4 (Unambiguity of schedule version)**.** *At any point in time, a schedule with version* v *is unique, i.e., if two nodes* $i, j \in [1, N]$ *simultaneously know a schedule with version* v*, then* $\forall k \in [1, K] : S_i^v[k] = S_j^v[k]$.

Proof. We prove this theorem by induction. As a base case, we use the first schedule S^1 with the unique initialization of the slot assignments $\forall k \in [1, K] : S^1[k] = \perp$ (line 40).

As an induction step, we assume that a unique schedule S^{v-1} has been generated based on a complete set of information (M'^{v-1}, R'^{v-1}), where $|\{j \in [1, N] : M'^{v-1}[j] = 1\}| > N/2$. As M'^{v-1} consists of a majority of nodes, we know that a subset of nodes from M'^{v-1} must be part of any complete set of information (M'^{v}, R'^{v}) in order to generate schedule S^v (as two majorities of nodes cannot be disjoint and input from a majority of nodes is required (line 21)). During each round, we know from Corollary 4.1 that a maximum of one complete set of information can be generated. As the membership flags remain fixed during an epoch (with lines 13 and 15 only being executed at the end of an epoch), this property also holds for the entire epoch. Therefore, we can conclude that maximally one unique schedule version v can be generated per epoch.

However, it remains to be seen whether the version can persist after a change of membership flags at the end of the epoch and will become the dominant schedule version. We know that a schedule is only prevented from expiration if a node receives information from a majority of nodes from its MSCC at least once during the epoch (line 5). If sufficient nodes are using the previous schedule S^{v-1} in a future epoch so that a new complete set of information $(M'^{v'}, R'^{v'})$ can be formed, the previously computed schedule S^v is not in use by a majority of nodes (otherwise, no majority of nodes using version $v-1$ could have been found to generate version v').

Therefore, nodes using S^v must belong to a different SCC and could only set $I_e[j] \leftarrow 1$ (line 17) for nodes $j \in \Delta M = \{i \in [1, N] : M'^{v'}[i] = 0\}$, because others cannot pass the check in line 15 as they belong to the SCC using $M'^{v'}$. As we know that $|\{j \in [1, N] : M'^{v'}[j] = 1\}| > N/2$, $|\{j \in [1, N] : \Delta M[j] = 1\}| < N/2$. Therefore, all nodes with schedule S^v must discard their schedule in line 6 at the latest at the end of the same epoch in which $S^{v'}$ has been generated. Hence, we can conclude that at any point in time, a schedule of version v is unique. □

Theorem 4.5 (Safe conduct of unsynchronized nodes)**.** *A network operating on the main channel does not experience interference during the* `DD` *phase due to unsynchronized nodes.*

Proof. A node that lost contact with its network $\mathcal{N}$ will have its schedule expire at the end of the next epoch (line 6) and will not be able to participate in line 2 during the `DD` phase thereafter. It can still try to regain connectivity and will communicate with other nodes during the `SN` phase for up to E_{max} epochs without influencing the `DD` phase, after which it will be forced to revert to bootstrapping (line 8). As mentioned in Section 3, E_{max} must be chosen appropriately so that the maximal clock drift within this time may not cause interference.

As Algorithm 1 is using the boot channel for all transmissions, no interference with the main channel can result from bootstrapping nodes. A node can only return to the main channel in one of two cases. In line 12, it can switch to the main channel in case it re-synchronizes to the network $\mathcal{N}$ on the main channel in line 4 and hence cannot interfere with the `DD` phase as it just synchronized and uses $v = 0$ (line 6). In line 43, a node can switch to the main channel as part of a majority of nodes that forms a new network $\mathcal{N}'$. As the previous network $\mathcal{N}$ can only remain operational as long as at least a majority of nodes regularly interact and execute line 17 to prevent schedule expiration in line 6, this indicates that only a minority remains on the main channel. For a node to switch in line 8, it must have lost connectivity for at least $E_{max} > 0$ epochs during which no majority from $\mathcal{N}$ was interacting. As a single epoch of missing interaction is sufficient for all these nodes to have their schedules expired, all remaining nodes of $\mathcal{N}$ must have executed line 6 at least once and therefore have $v = 0$. Therefore, the previous network $\mathcal{N}$ does not exchange packets anymore during the `DD` phase as well as the `SD` phase. The latter conclusion follows as a new schedule cannot be computed due to the condition in line 21 not being satisfied. With $\Delta\tau \geq F \cdot T$, all remaining nodes of $\mathcal{N}$ will have had to execute line 8 and have switched to the boot channel before $\mathcal{N}'$ finishes waiting in line 37, resulting in no more transmissions during an `SN` phase on the main channel. Therefore, when a new network $\mathcal{N}'$ starts communicating on the main channel, all other nodes must have switched to the boot channel and the network can operate without interference. □

4.2 Liveness

The following Theorem 4.6 shows that nodes continue participating using the current schedule and can communicate data packets in the `DD` phase of every round even under very general failure conditions.

Theorem 4.6 (Liveness of data dissemination)**.** *Every node participates in the* `DD` *phase unless both of the following conditions are satisfied: (i) It did not receive a schedule in any* `SD` *phase of the previous epoch and (ii) during the entire previous epoch, it only stored the requests from a minority of nodes.*

Proof. A node does not participate in the `DD` phase if $v = 0$, i.e., if line 6 was executed. This only happens if no more than $N/2$ nodes $j \in [1, N]$ satisfy $I_e[j] = 1$ (line 5). If the node has received a schedule, then $I_e[j] \leftarrow 1$ for all nodes (line 45) and the condition is not satisfied. If the node stores the request of a node j, it sets $I_e[j] \leftarrow 1$ (line 17). Therefore, if the node did store the requests from more than $N/2$ different nodes at least once during the epoch, the condition to set $v \leftarrow 0$ is not satisfied either. □

Note that a schedule is typically computed and distributed by many nodes due to the high level of redundancy. Therefore, a node will only fail to receive a schedule in any round of an epoch under a significant amount of very specific and simultaneous node or link failures. And even in such a case, a node may still participate in the data exchange if it received and stored requests from a majority of nodes at least once during the entire previous epoch. Through the use of the epoch length F, this aggregation period can be arbitrarily extended, which diminishes the chance of a loss of liveness due to short-term failures or disturbances.

4.3 Adaptivity

Theorem 4.7 (Perpetuality)**.** *Nodes using Hydra are always able to eventually adjust to a change in operating conditions such as demand or environmental influences.*

Proof. While failures are occurring, adaptivity might be consistently delayed due to new nodes joining and leaving. However, Hydra cannot find itself in a situation where no more failures are occurring but the nodes cannot adapt.

With Algorithm 1, any majority of nodes can eventually re-establish a network in line 43 once they could synchronize and exchange messages. Furthermore, a node that is able to communicate with a majority of nodes will eventually encounter their communication and join in line 12. As it can arbitrarily often execute line 14 and restart the algorithm, it will never get stuck during bootstrapping.

With Algorithm 2 and Algorithm 3, an established network is also guaranteed to eventually find consensus and progress if communication is possible. If a node or link has failed, the network membership will be updated in line 15 at all remaining nodes and consensus can be found again. If a majority cannot be established anymore due to insufficient nodes, nodes can return to bootstrapping in line 8 and attempt to re-combine with other sub-networks. If a node is missing the newest schedule, it can receive it in line 43 as it will be re-transmitted by nodes that have indicated that they know the maximum schedule version in the network. Once the preconditions for schedule computation in line 21 (a complete set of information from a majority of nodes) and line 22 (equal schedule versions) are met, the network can adapt its schedule. □

Theorem 4.8 (Support for scheduling algorithms)**.** *All deterministic real-time scheduling policies can be supported, i.e., the slot assignment S^v can be transformed into any slot assignment $S^{v'}$ in at most two steps.*

Proof. We distinguish three cases of slot transitions from one schedule to another. For $\{k \in [1,K] : S^v[k] = S^{v'}[k]\}$ (i.e., the slots whose assignment does not change from S^v to $S^{v'}$), no changes need to be made. For $\{k \in [1,K] : S^v[k] = \perp\}$, $S^{v'}[k]$ can be set to any desired value within one step as the assignment is not restricted in this case. For $\{k \in [1,K] : S^{v'}[k] = \perp\}$, assigned slots from $S^v[k]$ can be released within one step.

For the remaining set $\Delta S = \{k \in [1,K] : (S^v[k] \neq S^{v'}[k]) \wedge (S^v[k] \neq \perp) \wedge (S^{v'}[k] \neq \perp)\}$, we create an intermediate schedule $\tilde{v}$ as a first step where

$$S^{\tilde{v}}[k] = \begin{cases} \perp & \text{if } k \in \Delta S \\ S^v[k] & \text{else} \end{cases}$$

In a second step, we can use the same argument for the case of $S^{\tilde{v}}[k] = \perp$ to conclude that we can now also set $S^{v'}[k]$ to any desired value $\forall k \in \Delta S$. This shows that a maximum of two schedules is required to transform a slot assignment into any other desired slot assignment. Because the algorithm is deterministic by definition and the inputs to the schedule have been shown to be equivalent in Theorem 4.1, the de-centralized computation of any deterministic real-time scheduling policy in line 24 can be supported. □

5 Acronyms

SPOF single point of failure

CRC cyclic redundancy check

SCC strongly connected component