Executable Rounds: a Programming Abstraction for Fault-Tolerant Protocols

Cezara Dragoi, Josef Widder, Damien Zufferey

Executable Rounds: a Programming Abstraction for Fault-Tolerant Protocols

CEZARA DRĂGOI, INRIA, ENS, CNRS, PSL
JOSEF WIDDER, Interchain Foundation
DAMIEN ZUFFEREY, MPI Kaiserslautern

Fault-tolerant distributed systems are notoriously difficult to design and implement. Although programming languages for distributed systems is an active research area, appropriate synchronization primitives for fault-tolerance and group communication remains an important challenge. We present a new programming abstraction, *HSync*, for implementing benign and Byzantine distributed protocols. *HSync* is based on communication-closed rounds. Round models offer a simple abstraction for group communication and *communication-closed rounds* simplify dealing with faults. Protocols are implemented in a modular way in *HSync*. The language separates the message reception from the process local computation. It extends classic rounds with language constructs that give to the programmer the possibility to implement network and algorithm-specific policies for message reception. We have implemented an execution platform for *HSync* that runs on top of commodity hardware. We evaluate experimentally its performance, by comparing consensus implementations in *HSync* with LibPaxos3 and Bft-SMaRt, two consensus libraries tolerant to benign, resp. Byzantine faults.

1 INTRODUCTION

Fault-tolerant distributed algorithms are crucial for many applications with high reliability requirements. These algorithm are notorious for being hard to design and implement. One needs to take into account features of a non-deterministic network and behaviors of faulty processes intertwined with the asynchrony of the actions performed by correct processes. Nevertheless, they are implemented for more and more applications [Baudet et al. 2019; Buchman 2016; Junqueira et al. 2011; Kotla et al. 2009], typically in generic all purpose programming languages which facilitate bug occurrences [Jepsen 2018], as a suitable programming abstractions is still a research challenge.

Fault-tolerant distributed algorithms are designed by two different communities. The system community typically focuses on a specific algorithm and its implementation over a network in a general purpose programming language. In the distributed algorithms community, researchers strive to propose general computational models in which they analyze theoretical (i.e., unimplemented) algorithms. For instance, consider the consensus problem, i.e., n processes trying to agree on a value. The seminal papers [Castro and Liskov 2002; Oki and Liskov 1988] define a solution for consensus in the benign, resp. the Byzantine case, with all the implementation details, including checks on the network to ensure the assumptions they need in order to solve consensus hold. On the other hand the seminal paper [Dwork et al. 1988] introduces the *basic round model* and presents several solutions for consensus in both the benign and byzantine case, under the assumptions that the network is partially synchronous, providing no implementation of these solutions.

Programming languages are the only way to bridge the gap between these approaches. From a programming language perspective, the main challenge is to identify appropriate synchronization primitives for fault-tolerant systems. We propose a programming language called *HSync* whose programming concepts inherit synchronization primitives from the computational models algorithm designers use, and performance-relevant implementation aspects. The language is equipped with an

Authors' addresses: Cezara Drăgoi, INRIA, ENS, CNRS, PSL; Josef Widder, Interchain Foundation; Damien Zufferey, MPI Kaiserslautern.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

execution platform that is based on features extracted from the implementations from the systems community. The programming abstraction we propose is addressed to the algorithm designer who wants to prototype algorithms without worrying about network code and data management.

The Network vs. the Algorithm. In the research literature on distributed algorithms — both with theoretical focus and with practical, system oriented focus — algorithm designers highlight and distinguish two aspects of their solutions: One the one hand, the network aspect, that is, how messages are sent and received by the programs, and how they are assumed to be handled by the underlying hardware/layers (e.g., reliable or timely communication, FIFO, etc.). On the other hand, the algorithmic (or protocol) aspect that describes the local computation depending on the current local state and the received messages. However, there are different approaches to separate the network aspects from the algorithm aspects, that correspond to different ways the network is abstracted. We distinguish asynchronous and round-based models. In the former, roughly speaking, messages are received one-by-one and typically timeouts are used to detect absence of a message and ensure progress. In the round-based approaches, a higher level of abstraction is used. Computations are structured in a sequence of rounds, and within one round a set of messages is received. This model is more abstract as the message reception of a round is hidden from the algorithmic aspect. A non-deterministically chosen subset of the sent messages is received in one atomic step.

Round Models: Separating the Network from the Algorithm. Synchronous (or round-based) approaches starts from idealized round-based computational models [Charron-Bost and Schiper 2009; Dwork et al. 1988; Gafni 1998], which has the advantage of simpler behaviors which entails simpler designs and simpler correctness arguments [Aminof et al. 2018; Dragoi et al. 2014; Marić et al. 2017; Stoilkovska et al. 2019] due to the fundamental notion of communication closure [Chou and Gafni 1988; Elrad and Francez 1982]. This means that distributed computations are organized in iterations (called rounds) where within a round all processes exchange messages, and then change their local state. Messages are not received outside the round they were sent, making rounds a powerful synchronization primitive in the presence of faults. The disadvantage is that the gap from the idealized round-based semantics to the asynchronous code is considerable. In particular, round structure are known for their poor performance due to an excessive need of synchronization. Still, regarding the design process, even in the asynchronous approach, algorithm designers, when presenting their solutions, describe the algorithms in terms of rounds, phases, epochs, ballots, etc., e.g., [Dwork et al. 1988; Lamport 2005; Ongaro and Ousterhout 2014]. The reason is that these notions encode some logical time that allows to structure and decompose distributed computations along the time line. Although presented in rounds, the system is implemented under the asynchronous semantics because it is permissive to network-specific performance optimizations, that can still force the implementation to satisfy the minimum required level of synchrony.

Our goal is to simplify the round-based design and implementation approach, and close the gap to the asynchronous semantics of the underlying network. We present *HSync*, a programming abstraction that encapsulates the following key functionalities of distributed protocols:

Message accumulator. The code receives messages one-by-one, and encodes based on the received messages or timeouts, which messages to store in the mailbox for the current round and when to stop waiting for more messages.

Round-based algorithm. When enough messages are received or a time-out expired, a high-level protocol computation should be encoded, e.g., depending on the mailbox for the current round, compute the majority value of the messages received in that round. This is inherently round-based and communication-closed.

```
def init(io: TpcIO) = {
    callback = io
    transaction = io.trId }
val rounds = phase(
 new EventRound[Int] //Round: Propose transaction
   def send(): Map[ProcessID,Int] = {
     if (id == coord) broadcast(transaction)
     else Map.empty[ProcessID,Int]
   }
   def receive_init = {
        Progress.strictWaitMessages }
   def receive(sender: ProcessID, payload: Int) =
        { Progress.goAhead }
   def finishRound(mbox: List[(ProcessID, Int)])
      = { commit = check(local_state, mbox) }
 },
 new EventRound[Boolean]{//Round: Commit Yes/No
   def send(): Map[ProcessID,Boolean] = { Map(
        coord -> commit ) }
   def receive_init = {
      if (id != coord) Progress.goAhead
      else Progress.strictWaitMessage}
   def receive(sender: ProcessID, payload:
        Boolean) = {
     if(!payload || mbox.length == n)
          Progress.goAhead
     else Progress.unchanged}
   def finishRound(mbox:List[(ProcessID,
        Boolean)]) = {
     if (id == coord) { decision = head(mbox) }}
 }, // Round Commit/Rollback, Round Ack
)
```

Fig. 1. First rounds of *Two phase commit* in *HSync*.

These two functionalities are interfaced via *message predicates* that are formalized in LTL interpreted over rounds. They describe in an assume/guarantee manner, on the one hand, the guarantees of the message accumulator regarding the mailbox it provides to the round-based computation, and on the other hand, the expectations of the round-based computation.

Contributions. We propose a new programming abstraction, called *HSync*, embedded into the SCALA programming language. We are the first to propose a language that lies between the synchronous and asynchronous paradigms. Our main contributions are:

- *HSync* programs have a round-based structure which allows algorithm designers to specify both the network and the algorithm aspects. Thus, one can focus on high-level algorithm aspects, and optimize protocols without having to go into details of network code, etc.
- *HSync* is suitable for both benign and byzantine fault-tolerant protocols,
- *HSync* allows the algorithm designer to write custom code that locally controls the round boundaries, i.e., the message accumulator, expressing a wide range of protocols,
- *HSync* compiles to efficient asynchronous code, that can be executed over any asynchronous network. The runtime environment ensures a sound and efficient round structure for benign and byzantine faults, based on seminal work by Dwork et al. [1988].

2 OVERVIEW

In this section we present the main features of *HSync*, using well-known example protocols. A program in *HSync* is structured in rounds, which can be either benign rounds or byzantine rounds, depending on the program being tolerant or not to byzantine processes (processes that do not follow the protocol). A program is a sequence of benign or byzantine rounds, called *phase*, which is executed in lock-step in a loop. Lock-step rounds means that there is no interleaving between the actions performed by any processes in two different rounds. In a round, processes send and receive messages and update their local state (w.r.t. the set of received messages).

2.1 Two Phase Commit

Fig. 1 presents the first two rounds — i.e., the first phase — of the protocol *two phase commit* in *HSync*. It is an atomic commitment protocol, executed by n benign processes (that is all processes follow the protocol). The entire protocol is structured in four rounds. A coordinating process, denoted `coord` receives transactions from a client and tries to get them committed, one by one, at

all the other processes in the network, that implement a replicated database. The coordinator's identity is fixed and the transactions are received via an input-output object, `io:TpcIO` at line 3. Fig. 2(a) shows an execution of two phase commit, where after two back and forth communication steps (that is four rounds) the transactions are committed at all processes, and Fig. 2(b) shows an execution where the transaction is aborted. The first round, starts at line 5. The parameter `Int` is the payload type of the messages exchanged in that round.

In the example, the integer is a transaction identifier that the coordinator proposes (at line 7) to all processes. Each process checks locally whether it can locally commit the transaction (without violating the consistency of the database) at line 13. In the second round, starting at line 9, processes send (at line 16) their vote to the coordinator. If one process refuses the transaction, then the coordinator proposes to abort, otherwise to commit. This decision is calculated at line 24 based on the received set of messages. In the next two rounds, which we omit for brevity, the coordinator sends a commit or abort message to all and after it receives acknowledgments from all processes, it informs the client of the decision and restarts with a fresh transaction.

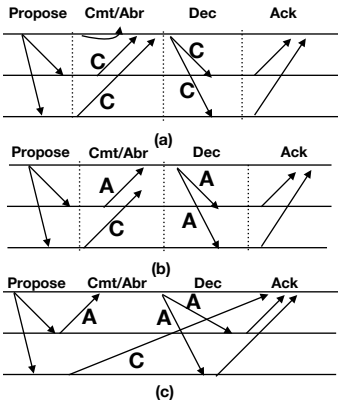


Fig. 2. Two-phase-commit: (a) and (b) are lock-step executions, (c) is a runtime execution.

In *HSync*, processes execute in lock-step a sequence of rounds called a `phase`. Each round has four methods: `send`, `receive_init`, `receive`, and `finishRound`. Within a round, processes execute first `send`, which defines the messages to be sent, followed by `receive_init` and multiple or no calls to `receive`, which compute the set of messages received by a process. Lastly, `finishRound` is executed, which defines how the process updates its state, e.g., in round 1, processes check locally if the transaction proposed by the coordinator can be committed, or in round 2, the coordinator updates the decision to commit or abort based on the set of received messages. The methods `send` and `finishRound` are executed synchronously by all processes, while `receive_init` and `receive` are executed asynchronously across processes (within the round boundaries). The methods `send` returns a map, indexed by the identity of the receivers and the values are the payloads. Although all messages are sent at once, we assume the protocol runs over a network that can lose or delay messages. Therefore, messages are received one by one until enough messages have been received and the method `finishRound` is called. The call to `finishRound` receives as input the list of received messages (sorted in the arrival order), called mailbox or `mbox` for short.

Message predicates and message accumulators. Two phase commit is a blocking protocol. It requires that in round 1 all processes receive a message (from the coordinator since it is the only process that sends), and in round 2 the coordinator must receive either a message from all processes or an "abort" message from some process. These properties on the set of received messages imposed by the algorithm we call *message predicates*. We formalize them in LTL (Linear Temporal Logic) over constraints on the mailbox which are evaluated when the rounds finish, e.g.,

$$\begin{gathered} \mathbf{G}\,\forall p.|mbox_p^1| > 0 \\ \mathbf{G}\,\forall p \exists q.p = coord \rightarrow \Big(|mbox_p^2| = n \vee head(mbox_p^2) = (q, false)\Big), \end{gathered} \tag{1}$$

```
1  var lastSeen = Map[ProcessID,Int]()
2  def getSuspected = {
3    lastSeen.filter{ case (_, last) =>
4      last > hysteresis }.keySet() }
5  def init(nope: Unit) = { //initially no suspicion
6    for (i <- 0 until n) { lastSeen(i) = 0 } }
7
8  val rounds = phase(
9  new EventRound[Set[ProcessID]]{
10   def send: Map[ProcessID,Set[ProcessID]] = {
11     broadcast(getSuspected)}
12   def receive_init = {
13     Progress.timeout(period)} })
```

```
17   def receive(sender: ProcessID, suspected:
           Set[ProcessID]) = { Progress.unchanged} })
18
19   def finishRound(didTimeout: Boolean, mbox:
           List[(ProcessID, Set[ProcessID])]) = {
20    for ( (k,v) <- lastSeen ) { lastSeen(k) = v + 1}
21    for ((sender,suspected) <- mbox){
22       lastSeen(sender) = 0
23       for (s <- suspected if (lastSeen(s) != 0) {
24         lastSeen(s) = hysteresis + 1}}} //suspect s
25     println("replica: "+id+" suspecting: " +
             getSuspected)
26   }}}}
```

Fig. 3. Strong Eventually Fault Detector *HSync*.

where p and q are processes, $\texttt{mbox}_p^1$, $\texttt{mbox}_p^2$ is the mailbox of process p in the first, resp. second, round, which is a list with $head(mbox_p^2)$ the first element. The temporal operators are interpreted over phases, where the formula under the operator **G** holds in every phase.

The message predicates must hold in any execution of the program. They are implemented by `receive_init` (lines 10 and 17) and `receive` (line 11 and 20). We call the implementation of these methods a *message accumulator*, because `receive` executes in a loop and each execution adds a message to the mailbox. The termination is controlled by progress instructions, e.g., `Progress.goAhead` which signals the end of waiting, and the control should move to `finishRound`. For example, the coordinator does not wait for any messages in the first round and uses `Progress.goAhead` in line 11. To force followers to wait for messages, the programmer uses `Progress.strictWaitMessages` in `receive_init`, which forces a least one execution of the method `receive`. Since the only sender is the coordinator, after receiving one message followers terminate the message accumulator. In the next round the coordinator must wait for n messages or an "abort" message. The instruction `Progress.unchanged` is used to keep the coordinator waiting for messages, i.e., keep making `receive` calls, until the condition guarding `Progress.goAhead` in line 21 is true.

The message accumulators of two-phase-commit implement the message predicates in (1) if the network does not lose messages and processes do not crash. It is well known that on any other network the protocol is blocking.

HSync has an efficient execution platform which does not implement synchronization barriers at the end of each round. The round switch is determined locally, by the termination of the message accumulator. Therefore at runtime, processes might be in different rounds, and the runtime simulates the round structure. Fig 2(c) shows the runtime execution corresponding to the execution in Fig 2(b). Since the transaction is aborted by the second process, the coordinator switches rounds before receiving the third message. Actually, this message gets delivered much later, when the coordinator is in the last round, when it does not matter anymore, as the coordinator decided to abort the transaction. We prove that clients do not distinguish between the runtime and the lock-step executions of *HSync*. From the client's perspective, the runtime execution in Fig 2(c) and the lock-step execution in Fig 2(b) are equivalent, the transaction is aborted in both.

2.2 Fault Detector

Failure detectors were introduced [Chandra and Toueg 1996] as auxiliary modules that continuously output an estimate of the crashed processes in the system, estimate used by another protocol, e.g., solving consensus in [Chandra and Toueg 1996]. The program in Fig. 3 implements an eventually strong failure detector in *HSync*. A process p suspects a process q to be faulty if in h consecutive

rounds, p does not receive a message from q. When process p suspects q, it broadcast this information. Any process that receives p's message will suspect q, unless it received a message from q in the same round. Initially no process is suspected. The protocol has one round that is executed repeatedly. Each round execution starts with processes broadcasting the set of processes they suspect (initially empty). In `finishRound` the process updates its set of suspected processes, based on the received messages in `mbox` for the current round.

Fig. 4 shows an execution of the protocol in Fig. 3, where $p1$ is the faulty process, suspected by $p2$ after three rounds ($h = 3$). The messages of $p1$ are received by $p3$ in the first and second round but not by $p2$. In each round $p2$ increases `lastseen` of $p1$, counting the rounds since a message from $p1$ was last received (line 20). In the third round $p2$ suspects $p1$ to be faulty in line 4, because `lastseen` of p2 is greater than h. Therefore, $p2$ broadcasts its suspicion about $p1$, which reaches $p3$. Consequently, $p3$ suspects $p1$ in line 24, since it did not receive any message from $p1$ in the current round.

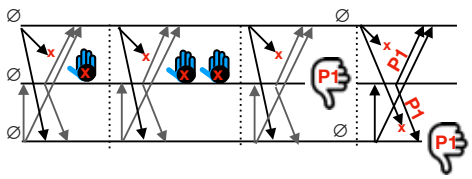


Fig. 4. An execution of the fault-detector in Fig. 3.

Message predicates and message accumulators. In order to ensure that faults are detected, and some correct process from some time on is not falsely suspected of being faulty, all executions must satisfy the following *message predicates*: (1) there exists a correct process p and all correct processes receive its messages, i.e., $\mathbf{F}\,\mathbf{G}\,\forall q \in correct.\ p \in mbox_q$, (2) no correct process receives a message from an earlier crashed process, $\mathbf{G}\,\forall p \in crashed, q \in correct.\ p \notin mbox_q$. There is no implementation of these message predicates unless the network has at least one correct process during the entire execution and the set of crashed processes is monotonically non-decreasing.

The message accumulator in Fig 3 implements the message predicates using timeouts. Each processes waits for messages up to a timeout. The timeout semantics is defined in `receive_init` by the instruction `Progress.timeout(period)` at line 13. Next, `receive` is executed in a loop, populating the round's mailbox, while the value of the timeout is decreasing. Contrary to the previous example, there is no `Progress.goAhead` in the code. The semantics of `Progress.timeout` states that when the timeout of the current round expires, processes execute `finishRound`. Waiting up to a timeout gives the possibility of more messages to be received (less chances of mistakenly suspect a process). However, an upper bound on the waiting time is necessary in order to avoid blocking, because many processes can be faulty.

The runtime executions are similar with the lock-step executions (assuming bounded clock drift), because all process wait up to the same timeout value before doing a round switch. This value is an input of the execution platform. If this timeout value is at least the message delay between the correct process and the rest of the network, then message accumulator implements the message predicates above, and the algorithm implements a failure detector.

2.3 ViewChange

View Change is a protocol used in the seminal paper on Practical Byzantine Fault Tolerance (PBFT) [Castro and Liskov 2002]. ViewChange elects a leader using a quorum, and the new leader makes sure that all the replicas in its quorum agree on a history and on a *view*. The implementation of ViewChange in *HSync* is given in Fig. 6. For simplicity we omit the history and the state of a replica is a simple integer `val` representing the current view the replica is in. The protocol consists of a phase with three rounds, which are executed repeatedly. The implementation assumes

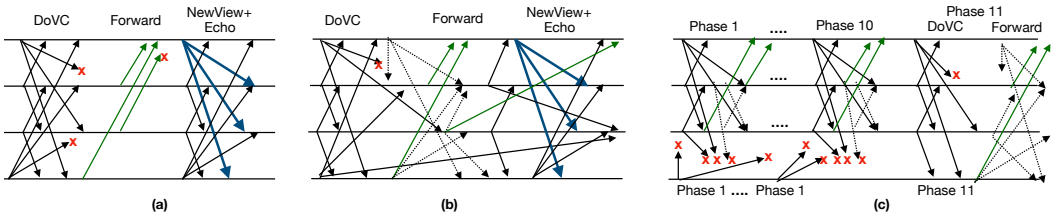


Fig. 5. Lock-step and runtime execution of View Change in Fig. 6.

authenticated communication channels between any two processes, that is, the receiver of a message can be sure which process sent the message (that is, *no masquerading*).

Fig. 5(a) shows an execution of ViewChange. In the first round all replicas broadcast their values (all-to-all communication) in line 4. In the byzantine case, receiving one message is not sufficient to trust the received value, because byzantine processes might send different (possibly faulty) values to different processes. Therefore, in the second round processes forward to the coordinator acknowledgements for all the received values in line 11 (where DoVc_mbox stores the pairs (v_i, p_i) received in the previous round). The ID of the coordinator is a function of the phase number (rotating coordinator). The coordinator trusts a value v_i sent by process p_i, if $2n/3$ processes acknowledge that they received v_i from p_i. If the coordinator receives $2n/3$ trusted values, based on them it computes the most recent state of the system. The function compute (at line 15) returns $\perp$ if the coordinator did not received $2n/3$ trusted values. In the third round the coordinator broadcast (line 20) the result of compute and the set of trusted pairs (p_i, v_i) that the computation was based on. At line 20, the other processes echo the acknowledgements they send to the coordinator (in the previous round) to the entire network. A process p that receives the coordinator's message, checks that the message was correctly computed — using $n/3$ messages by other replicas — in line 24. This holds either if any pair (v_i, p_i) received from the coordinator (as support for its decision) was received by the process p in the first round, that is, the pair is in DoVC_mbox, or if in the current round the process p received $2n/3$ echo messages certifying the v_i was sent by p_i.

```
1  val rounds = phase(
2    new ByzantineRound[Int]{//Round: DoVC
3      def send(): Map[ProcessID,Int] = {
4        broadcast(val)}
5      def finishRound(didTimeout:Boolean, box:
           List[(ProcessID, Int)]) = {
6        if(mbox.lenght > 2n/3)
7          DoVC_mbox = mbox }
8    },
9    new ByzantineRound[Boolean]{//Round: Forward
10     def send(): Map[ProcessID, List[Int]] = {
11       Map( coord -> DoVC_mbox )}
12     def finishRound(didTimeout:Boolean,
           mbox:List[(ProcessID, Boolean)]) = {
13       if (id == coord && mbox.length > 2n/3) {
14         decision = compute(mbox)
15       }}
16   },
17   new ByzantineRound[Boolean]{//Round: NewView
18     def send(): Map[ProcessID, List[Int]] = {
19       if(id == coord) broadcast(decision)
20       else broadcast (DoVC_mbox)}
21     def finishRound(didTimeout:Boolean,
           mbox:List[(ProcessID, Boolean)]) = {
22       if(mbox.length > n/3 &&
23            mbox.find(_.1 == coord ) )
24        decision = check_computation(mbox)}
25       out(decision)
26 }})
```

Fig. 6. View Change (PBFT) with authentication in *HSync*.

Message predicates. The algorithm is safe if less than a third of the processes is faulty and satisfies liveness, if eventually there is a good period where the quorum can communicate within itself and

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{interface\ body} \\
\mathit{body} & ::= & \mathit{var\_decl}^*\ \mathit{pure\_fun}^*\ \mathit{init\ phase} \\
\mathit{phase} & ::= & \mathit{round}^+ \\
\mathit{round} & ::= & \mathsf{EventRound}_{\,\mathsf{M}} \\
 & | & \mathsf{ByzantineRound}_{\,\mathsf{M}}
\end{array}
$$

Fig. 7. *HSync* abstract syntax.

$\mathsf{EventRound_M}$
- `mbox`: $\mathit{List}[\mathsf{Pid} \times \mathsf{M}]$
- `to`: $\mathbb{N} \cup \infty$
- `send`: $() \to [\mathsf{Pid} \mapsto \mathsf{M}]$
- `init`: $() \to \mathbb{N} \cup \infty$
- `receive`: $(\mathit{sender} : \mathsf{Pid},\ \mathit{payload} : \mathsf{M}) \to \mathbb{N} \cup \infty$
- `finishRound`: $(\mathit{mbox} : \mathit{List}[\mathsf{Pid} \times \mathsf{M}]) \to ()$

Fig. 8. Internal structure of rounds. EventRound and ByzantineRound are similar.

with the coordinator, that is, if $\mathbf{G}\ |\mathit{correct}| > 2n/3$, and

$$\mathbf{F}\ \exists Q \forall q \in Q.\ |Q| > 2n/3 \wedge Q \subseteq \mathit{correct} \wedge \mathit{coord} \in \mathit{correct} \\ \wedge\ Q \cup \{\mathit{coord}\} \subseteq \mathit{mbox}^1_q \wedge Q \cup \{\mathit{coord}\} \subseteq \mathit{mbox}^2_{\mathit{coord}} \wedge Q \cup \{\mathit{coord}\} \subseteq \mathit{mbox}^3_q.$$

The predicate is an instance of a more general predicate, called *partial synchrony*, stating that eventually all correct processes can talk to each other. This predicate, in its more general version, is implemented by default by the runtime for all so-called *closed rounds*, that is, when the programmer does not implement a message accumulator so that the default *HSync* message accumulator is used.

Fig. 5(b) shows a runtime executions of ViewChange in *HSync*. The implementation of the message predicate required for liveness forces the runtime to keep $n/3$ correct processes in sync in every round. It it well understood that otherwise there is no way to ensure progress after a bad period of time [Dwork et al. 1988]. Therefore, at runtime, more than $2n/3$ messages are required to switch rounds: as there are at most $n/3$ faulty processes, more than $n/3$ of these messages originate from correct processes. This is implemented ensuring an all-to-all communication (all processes broadcast) in all rounds. If not all protocol rounds are all-to-all (e.g., round "Forward"), the runtime adds extra messages for these rounds, in order to ensure $2n/3$ messages can be received, even if faulty processes do not send messages. These extra messages are shown with dotted edges in Fig. 5(b). Note that at runtime, messages from different rounds are not received in order. The runtime discards the late messages and store the messages from the future rounds. The runtime implements a catch-up mechanism that allows processes to jump to a future round, provided that there are more than n/3 processes that send messages for that round (or a higher one). Fig. 5(c) shows an execution where $p3$ was slow for a few rounds, but when it receives $n/3$ from the same round it jump to it and joins the other processes.

3 *HSync*: A DOMAIN-SPECIFIC LANGUAGE FOR FAULT-TOLERANT PROTOCOLS

We give syntax and its lock-step semantics. The semantics has synchronized transitions at the beginning and end of each round. However, the communication within a round is an interleaving of steps by the different processes and the network.

3.1 Syntax

The syntax of a *HSync* program is given in Fig. 7. A program defines the code executed by one process. In *HSync* all processes execute the same code. The sequence of rounds need to be the same across all the processes. However, within a round one may still encode different roles (e.g., coordinator, follower) by branching. The branching conditions may be over the process identifier or received messages (which can vary non-deterministically across processes).

A program is defined by an interface and the program's body. The *interface* is an object with a list of methods. A program implemented in *HSync* can be seen as a service which uses the methods

in the interface to communicate with external distributed programs, also referred to as clients of the service. There are inbound operations in the interface, used for example to process a new client request, and outbound operations (callbacks), called to deliver some results to the client, e.g., acknowledgments that the request has been processed.

The *program's body* consists in variables declaration, an initialization part followed by a computation part. All variables are local to a process. The communication between processes is done exclusively by message passing.

3.1.1 Variable declaration. The variables declared in the beginning of the program are called *algorithm variables*, denoted `Vars`. Their scope is the entire program. Each *HSync* program has also several predefined variables n the number of processes, f the number of byzantine processes (both parameters of the program), and r the round number. Rounds also have their own variables (we defer the discussion until rounds are introduced). Also, as usual variable declaration is allowed in any function of the program and their scope is limited to that function's body.

3.1.2 Initialization `init`*.* The initial values for the algorithm's variables, are defined using the inbound interface operations. The method `init` implements the initialization using sequential code and the interface inbound operations. The method `init` takes as input interface objects, it does not return any values, and modifies the algorithm's variables without sending or receiving messages.

3.1.3 Rounds. The computation part is structured into a fixed sequence of rounds, called `phase`. *HSync* has two different types of rounds: EventRound and ByzantineRound. that are sub-classes of the abstract classes Their structure is described in Fig. 8. Rounds are parameterized by a message type M and have two attributes `mbox` of type *List*[(Pid, M)] is a list of the messages received, where Pid is the process identity type, and timeout, `to` for short, a potentially infinite integer value. Rounds contain a few methods which define their behaviors. The methods `send` and `finishRound` must be defined by the program. An implementation of the methods `receive_init` and `receive` is optional. We discuss what the algorithm designer can implement in these methods below; if the designer implements `receive_init` and `receive`, the round is called an *open round*, otherwise it is called a *closed round*.

Syntactically both EventRound and ByzantineRound rounds are defined by the same methods, and their lock-step semantics is the same. However, the execution platform of *HSync* distinguishes the two kinds of rounds. In the presence of Byzantine faults, the platform performs a more involved round-synchronization algorithm. A program for Byzantine faults should use byzantine rounds. (Byzantine faulty processes may send different faulty round numbers to different peers which may lead to situations from which the simpler round-synchronization algorithm cannot recover.)

The type of messages exchanged in different rounds might differ, however within a round, all messages have the same payload type, denoted by M. The control flow of a round is a sequence of method calls, starting with `send`, followed by `receive_init` and a loop of calls to `receive`, if the latter two are present, and lastly `finishRound`. The round variables `mbox` and `to` are visible in all methods of the same round.

Sending messages. The method `send` does not take any inputs, and returns a partial map from process identity to a payload type, i.e., [Pid $\mapsto$ M], which associates receivers with payload values, i.e., the sent messages. The method `send` is side-effect free w.r.t. the algorithm variables.

Message accumulator. A message accumulator collects the messages delivered by the network in a round (to the process that executes it), up to the moment when a certain condition holds. The message accumulator stores the collected messages in the round variable `mbox`.

The algorithm designer implements the message accumulator by redefining `receive_init` and `receive`. Within these methods, the designer controls when the accumulator should terminate using different *progress instruction*:

- `Progress.strictWaitMessages` states that the message accumulator terminates when a condition on the received list of messages holds. This condition is defined in the method `receive` (e.g., Fig. 1 line 10). This instruction corresponds to an infinite timeout.
- `Progress.timeout(t)` states that the accumulator should terminate t time units after the round started (e.g., Fig. 3 line 13).
- `Progress.goAhead` marks the immediate termination of the message accumulator. If used in `receive_init` it means the accumulator terminates without collecting any messages, i.e., `mbox` $= \emptyset$ (e.g., Fig. 1 line 21). This instruction corresponds to a timeout of 0.
- `Progress.unchanged` maintains the previously set progress definition.

The function `receive_init` defines in initial progress instruction. Different processes may have different termination conditions of the message accumulator in the same round. For example, in the first round of two-phase-commit the leader's accumulator waits for messages while the followers accumulator terminates with an empty `mbox`.

The function `receive` updates the state of the message accumulator for the current round. `receive` takes as input an incoming message, that is, a pair $(sender, value)$, where $sender$ is a value of process identity type `Pid`, and $value$ a value of the round's the payload type. Each execution of `receive` adds the input pair $(sender, value)$ to the `mbox` of the current round. When `Progress.goAhead` is reached or the timeout expires, the control moves to the function `finishRound`.

The message accumulator must be side-effect free w.r.t. the algorithm variables. The implementation of the message accumulator is optional. When methods `receive_init` and `receive` are not implemented, the mailbox non-deterministically defined (as we discuss in the semantics section).

Round-based algorithm. By implementing the method `finishRound`, the algorithm designer specifies how the algorithm variables are changed at the end of a round. The method `finishRound` takes as input the queue of messages received, i.e., takes as input `mbox`. `finishRound` is a sequence of instructions that may call outbound operations from the interface, to communicate the results of its computation to a client.

3.2 Lock-step semantics of *HSync*

Given a program $\mathcal{P}$, all process execute, in a loop, the sequence of rounds defined in the `phase` array of $\mathcal{P}$. We call *phase* one loop iteration. The semantics of a program is given by the synchronized parallel composition of the transition systems associated with each process.

The execution progresses in lock-step, that is, all processes execute the same round in the same time. Within a round, the methods `send` and `finishRound` are executed synchronously by all processes, while the executions of `receive_init`, `receive`, and the network transitions are interleaved across processes within the round boundaries. Locally, processes execute the same sequence of calls to round methods, i.e.,

$$round\_execution ::= init;\ send\big((receive\_init;\ receive^*) \vee NondetNet\big)\ ;finishRound,$$

where *NondetNet* represents a network transition.

The state S of a *HSync* program is represented by the tuple $\langle gstep, s, r, msg\rangle$ where:

- $gstep \in \{Send, Recv, Net, Fin\}$ indicates whether the next method is `send`, `receive`, a network transition, or `finishRound`, respectively;
- $s \in [P \rightarrow V \cup \{lstep\} \rightarrow \mathcal{D}]$ stores the local states of the processes, where $lstep \in \{Send, Recv, Net, Fin\}$ and V is the union of all variables;

Synchronous transitions

$$\textsc{Init}\quad \frac{\forall p \in Pr.\ * \overset{\mathtt{init}(),o_p}{\Longrightarrow} s(p) \quad O = \{o_p \mid p \in Pr\}}{* \overset{\{\mathtt{init}_p()\mid p\in Pr\},O}{\Longrightarrow} \langle Send, s, 0, \emptyset, \emptyset\rangle}$$

$$\textsc{Send-with-Recv}\quad \frac{\forall p \in Pr.\ s(p) \overset{m_p=\mathsf{phase}[r].\mathsf{send}()\in[P\rightharpoonup M]}{\Longrightarrow} s(p) \quad sms = \{(p, m, q) \mid q \in P \wedge m \in M \wedge (q, m) \in m_p\}}{\langle Send, s, r, \emptyset, \emptyset\rangle \overset{\emptyset,\{m_p=\mathsf{send}_p()\mid p\in P\},\emptyset}{\Longrightarrow} \langle Recv, s, r, sms, \emptyset\rangle}$$

$$\textsc{Send-no-Recv}\quad \frac{\forall p \in Pr.\ s(p) \overset{m_p=\mathsf{phase}[r].\mathsf{send}()\in[P\rightharpoonup M]}{\Longrightarrow} s(p) \quad sms = \{(p, m, q) \mid q \in P \wedge m \in M \wedge (q, m) \in m_p\}}{\langle Send, s, r, \emptyset, \emptyset\rangle \overset{\emptyset,\{m_p=\mathsf{send}_p()\mid p\in P\},\emptyset}{\Longrightarrow} \langle Net, s, r, sms, \emptyset\rangle}$$

$$\textsc{RecvStart}\quad \frac{\forall p \in Pr.\ s(p)|_{\mathsf{vars}} = s'(p)|_{\mathsf{vars}} \wedge s'(p)(lstep) = Recv}{\langle Recv, s, r, sms, dms\rangle \overset{\emptyset,\emptyset,\emptyset}{\Longrightarrow} \langle Recv, s', r, sms, dms\rangle}$$

$$\textsc{RecvEnd}\quad \frac{\forall p \in Pr.\ s(p)(lstep) = Fin}{\langle Recv, s, r, sms, dms\rangle \overset{\emptyset,\emptyset,\emptyset}{\Longrightarrow} \langle Fin, s, r, sms, dms\rangle}$$

$$\textsc{FinishRound}\quad \frac{\forall p \in Pr.\ s(p) \overset{\mathsf{phase}[r].\mathtt{finishRound}(\mathsf{mbox}_p),o_p}{\Longrightarrow} s'(p) \quad s'(p)(\mathsf{mbox}_p) = [] \quad r' = r + 1 \quad O = \{o_p \mid p \in Pr\}}{\langle Fin, s, r, sms, dms\rangle \overset{\emptyset,\{\mathtt{finishRound}_p(\mathsf{mbox}_p)\mid p\in Pr\},O}{\Longrightarrow} \langle Send, s', r', \emptyset, \emptyset\rangle}$$

Nondeterministic network transitions for closed rounds

$$\textsc{NoRecv-NetworkSteps}\quad \frac{(sms, dms) \overset{N,\emptyset,\emptyset}{\Longrightarrow} (sms', dms') \quad N \in \{Dv, Dr, Cr\}}{\langle Net, s, r, sms, dms\rangle \overset{N,\emptyset,\emptyset}{\Longrightarrow} \langle Net, s, r, sms', dms'\rangle}$$

$$\textsc{NoRecv-Mailbox}\quad \frac{\begin{array}{l}\forall p \in Pr.\ s(p)(lstep) = Net \wedge s'(p)(lstep) = Fin\\ \wedge s(p)|_{\mathsf{vars}} = s'(p)|_{\mathsf{vars}} \wedge s'(p)(\mathsf{mbox}_p) = dms(p)\end{array}}{\langle Net, s, r, sms, dms\rangle \overset{\emptyset,\emptyset,\emptyset}{\Longrightarrow} \langle Fin, s', r, sms, dms\rangle}$$

Fig. 9. The global semantics of *HSync*. Cr is the set of crashed processes, i.e., $Cr = \{p \in P \mid s(p) = \bot\}$. Crashed processes never recover, i.e., $\forall p \in Cr.s(p) = s'(p)$. Pr is the set of running processes, i.e., $Pr = P \setminus Cr$. A transition $s(p) \overset{\mathsf{phase}[r].m,o}{\longrightarrow} s'(p)$ denotes that p executes, in local state $s(p)$, the method m of the EventRound $\mathsf{phase}[r]$. The execution produces observable events o, corresponding to calls to methods from the interface.

- $r \in \mathbb{N}$ is a counter for the executed rounds;
- $sms \subseteq 2^{P,M,P}$ is a set of sent messages in a round and $dms \subseteq 2^{P,M,P}$ is a partially ordered set of delivered messages, such that the messages delivered to each process are totally ordered.

The most important transitions of the semantics of a *HSync* program are shown in Fig. 9 and Fig. 10. A transition is written as $S \overset{N,I,O}{\longrightarrow} S'$ where S, S' are states, N is a set of labels that identify network transitions, O is a set of labels defined from the names of the methods in the interface. I is a set of labels not in the interface corresponding to internal transitions. A client of a *HSync* program can only observe the transition labels in O.

3.2.1 Init. Initially the state of the system is undefined, denoted by $*$. Rule Init in Fig. 9 is the first transition of every process p, corresponding to calling `init`. This transition may use inbound operations from the interface to get requests from a client, denoted o_p, where o is the operation's name and p is the process executing it. The `init` method does not return a value but initializes the state of the system, and moves the control to *Send*. Initially, the round counter is 0 and there are no messages in the system. Rounds are executed in a loop, in the order defined by the `phase` array, and the round number is incremented at the end of each round.

3.2.2 Send. The first transition of a round is Send, which adds the messages sent by processes to the pool of sent messages, *sms*. The messages in *sms* are triples of the form (sender, payload, recipient),

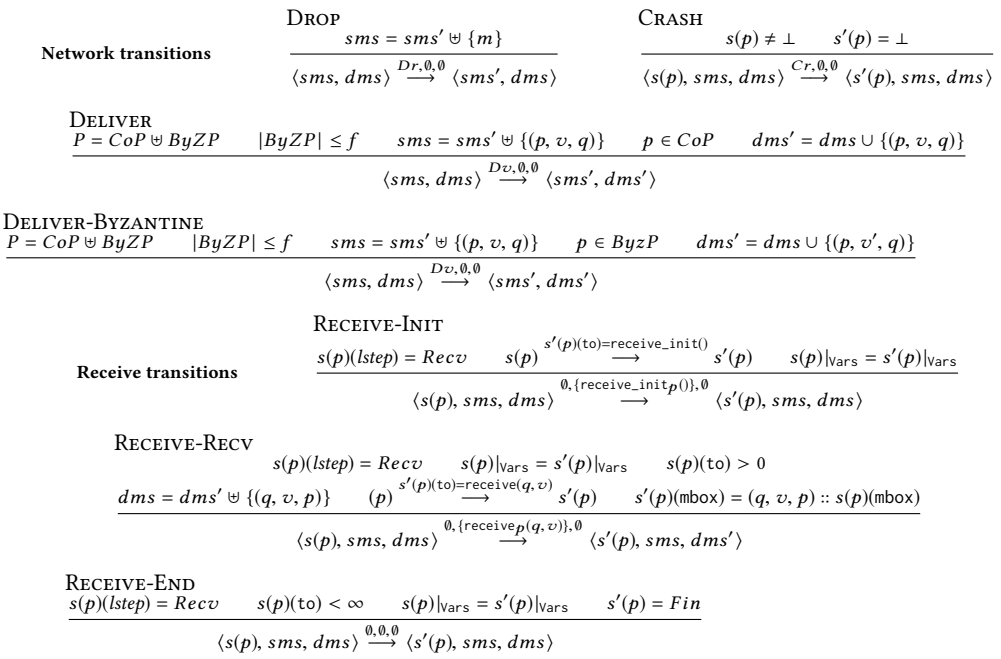


Fig. 10. Message accumulator transitions.

where the sender and receiver are processes and the payload has type M. The triples are obtained from the map returned by send to which we add the identity of the process that executed send. The send operation does not affect the state of the processes. All processes execute synchronous (without any interference) the send method of the current round. The semantics considers two send rules. Send-with-Recv is applied when `init` and `receive` are redefined. The control label *gstep* becomes *Recv*, i.e., the control goes to the implementation of a message accumulator. Send-no-Recv is applied when `init` and `receive` have the default implementation and in this case the control goes to the network, that is *gstep* equals *Net*.

3.2.3 Network transitions. The pool of messages *sms* and the lists of delivered messages *dms* are managed by a special *network* process whose only local variables are *sms* and *dms* (for simplicity the rules omit its identity). The rule Deliver, in Fig. 10, captures how the network takes a message m from the message pool *sms* and puts it into the reception list of the message's receiver, i.e., $dms(m.receiver)$. For the byzantine case, we assume that all programs run over a network that allows at most f byzantine processes, in some set *ByzP*, and these messages can be altered before delivered. The rule DROP captures how the network can drop sent messages by removing them from the send pool *sms*. Processes may crash (see Crash) and never recover.

3.2.4 Message reception in closed rounds. When `receive_init` and `receive` are missing (they are not redefined), the rule NoRecv-NetworkSteps is applied, a nondeterministic number of times, followed by the rule NoRecv-Mailbox, from Fig. 9. The mailbox of each process is set to be equal with the content of the delivery queue of the process. It is totally under the control of the network, that populates `mbox` with a non-deterministic chosen (possibly altered) subset of the sent messages.

3.2.5 Message accumulator: Message reception in open rounds. The implementation of the message accumulator starts by updating the local variable *lstep* to *Recv* (see rule RecvStart in Fig. 10). Each process executes first `receive_init` (see rule Receive-Init in Fig. 10) and afterwards a loop of calls to `receive` (see rule Receive-Recv in Fig. 10). Calls to `receive` stop when either the instruction `Progress.goAhead` is reached or when a timeout expired.

Accumulator transition Receive-Init. The method `receive_init` is used to control the waiting for messages in the current round. If `receive_init` reaches `Progress.goAhead` no `receive` is executed in that round. If `receive_init` reaches `Progress.strictWaitMessages`, a round will repeatedly execute `receive` until `receive` reaches a progress instruction that terminates the message accumulator. If `receive_init` reaches `Progress.timeout(t)`, it sets the timeout `to` to a non-deterministic value. Timeout is captured by an the round variable `to`, whose initial value is either defined by `receive_init`, or it is ∞. If `to` is initialized by `receive_init`, then when the timeout expires, no more calls to `receive` are executed and the execution of `finishRound` is enabled (i.e., $lstep = Fin$).

Accumulator transition Receive-Recv. The method `receive` when executed by process p, takes as input a delivered message in the queue $dms(p)$ (see rule Receive-Recv). The algorithm variables are not modified by `receive`. If no messages are available and the process is not waiting (`to` $= \infty$), it can change get ready for `finishRound` (see rule Receive-End).

Progress instructions. The accumulator transitions `receive_init` and `receive` use progress instructions to control the waiting for messages. The instruction `Progress.goAhead` sets the round's timeout to 0 and sets *lstep* to *Fin*, stopping the execution of the sequence of receives. The instruction `Progress.strictWaitMessages` sets the timeout to ∞, while `Progress.timeout(t)` sets the timeout to a non-deterministic value. The result of the instruction `Progress.unchanged` depends on the context. If a timeout is active, then `Progress.unchanged` decrements the timeout, otherwise it does not change any variables.

The *accumulator transitions* are interleaved with the *network transitions* defined in Fig. 9. While collecting messages, the local variable *lstep* equals *Recv* until it gets updated to *Fin* at the end of the message accumulator. Globally the systems transitions into a *Fin* global state when all non-crashed processes have their local variable *lstep* equal to *Fin*, rule RecvEnd in Fig. 9.

3.2.6 Round-based Algorithm transition. In a FinishRound transition (Fig. 9), all processes execute synchronously (without any interference) the method `finishRound` of the current round. Locally, on each process p, the set of received messages $\texttt{mbox}_p$ (computed previously) is the input of `finishRound`. The `finishRound` operation might produce an observable transition o_p. At the end of the round, *sms* and *gmsg* are purged and r is incremented by 1.

4 FORMALIZING NETWORK GUARANTEES

The semantics defined above imposes no restrictions on which or how many messages are received. It includes executions where no messages are received (in time), which corresponds to general asynchronous fault-prone networks. Many distributed computing problems, e.g., agreement or consensus, are not solvable under such network assumptions [Fischer et al. 1985]. Therefore, algorithm designers consider restricted classes of networks, that deliver the messages required to solve the problem of interest. In *HSync* these restrictions are formalized as *message predicates* and are expressed in linear temporal logic LTL.

4.1 All-network semantics

Here, we introduce a notation for the most general semantics of a *HSync* program that considers no restrictions on the network definition from Fig. 9 and Fig. 10, and call it *all-networks semantics*: The *all-networks semantics* of a *HSync* program $\mathcal{P}$ is the set of execution defined by the transition system in Fig. 9 and Fig. 10, denoted by $[\![\mathcal{P}]\!]_{All-Net}$.

All executions in Fig. 2, 4, and 5 belong to the all-network semantics.

4.2 Message predicates

Message predicates are formulas in LTL over propositions that describe the global state of a *HSync* protocol at the end of the message accumulator, that is, before the `finishRound` transition. More precisely, propositions describe sets of messages, cardinalities, and message payloads. Given a program $\mathcal{P}$ and a message predicate *MP*, *MP* is interpreted over the all-network executions of $\mathcal{P}$. The LTL temporal operators are interpreted over phase (sequence of rounds) in the *HSync* semantics, that is, **G** means in every phase, **F** eventually, **X** refers to the next phase.

To describe the message predicates, from the network semantics in Fig. 10 we use the set of crashed processes (*Cr*), which is the set of processes whose state equals $\perp$, and the set of byzantine processes *BzyP*, which is a subset of processes, whose identity is not known. We define the correct processes as the set of processes that are neither crashed nor Byzantine, $correct = P \setminus (ByzP \cup Cr)$.

Definition 4.1 (MP-network executions). Given a *HSync* program $\mathcal{P}$ parametrized by a network predicate *MP*, the semantics of $\mathcal{P}[MP]$, denoted $[\![\mathcal{P}[MP]]\!]$ is the set of executions in $[\![\mathcal{P}]\!]_{All-Net}$ that satisfy the predicate *MP*, that is $[\![\mathcal{P}[MP]]\!] = \{\sigma \in [\![\mathcal{P}]\!]_{All-Net} \mid \sigma \models MP\}$. For uniformity of notations we consider $[\![\mathcal{P}[true]]\!] = [\![\mathcal{P}]\!]_{All-Net}$.

4.3 Partial synchrony: the default *HSync* message predicate

The *HSync* execution platform is based on the seminal work on partial synchrony by Dwork et al. [1988] and the generalized partially synchronous model by Chandra and Toueg [1996]. They consider networks with bounded message delays and bounded relative processing speeds. However, these bounds are unknown and only hold after some unknown global stabilization time. These assumptions model finite bad periods where the system behaves unpredictably because of high load, etc. The round synchronization algorithms by Dwork et al. [1988] ensure for such networks, that from some round on, every message that is sent by a correct process to a correct process is received in the same round it was sent. Formally,

$$PS := \mathbf{F}\,\mathbf{G}\,\forall p, q \in correct.\, p \in mbox_q^{round}$$

We provide the correctness arguments for the *HSync* execution platform in Section 5.3, which show that the platform indeed ensures the above predicate *PS* for closed rounds.

4.4 Custom message predicates

If the algorithm designer chooses to implement the message accumulator, the message predicate *MP* constitutes the interface between the message accumulator and the round-based algorithm. The round based algorithm can be designed under the assumption that *MP* holds, while it has to be ensured that the message accumulator implements *MP*.

In Section 2 we give examples for custom message predicates for two-phase commit and the failure detector implementation. Other examples of custom implementations of *PS* for specific networks are given in Sec. 5.4.

5 RUNTIME FOR *HSync*

We introduce an execution platform (or runtime) for *HSync* that works in asynchronous and partially synchronous networks. The executions generated by the runtime are refinements of the executions in the semantics of *HSync* from Section 3. In general, if the network does not provide any guarantees regarding reliability and timeliness of message passing, the runtime, also cannot guarantee non-trivial message predicates. However, for closed rounds, if the underlying networks ensure that from time to time there are good periods (partial synchrony), the runtime resynchronizes automatically and ensures that correct processes can communicate with each others reliably. If the message accumulator is implemented (open round) the runtime does not provide guarantees, it is in the responsibility of the designer to ensure/verify the required message predicates in the target network. We present design principles for such scenarios in Section 5.4.

5.1 Partially synchronous networks

We define the network over which *HSync* programs are executed. For a given message m, we denote by $T(m)$ the time it takes between the moment a message is added to the send pool *smsg* and the moment it either gets dropped by the network or it is delivered to its recipient. We define a sub-class of asynchronous networks, called *partially synchronous* that allow stronger compilation guarantees. It is a variant of the seminal model by Dwork et al. [1988].

A network is *partially synchronous* if there exists a Δ, and a global synchronization time *GST*, such that the transmission delays of all messages m sent after *GST* from a correct processes to a correct process satisfy $T(m) < \Delta$.

5.2 Runtime semantics

The challenge for the runtime is that the resulting asynchronous executions should be indistinguishable to the *HSync* semantics. The latter having globally synchronized round switches and communication-closed rounds. Due to interleaving, we cannot enforce synchronized round switches, so that different processes may be in different round at the same time. In addition, due to message reordering, the local runtime code may receive messages from future and past rounds.

Benign HSync programs. We present first the execution platform for benign *HSync* programs and below the byzantine case. The *runtime algorithm* in Listing 1 defines the code executed at runtime by one process. It is a wrapper for *HSync* code. The executions of a wrapped *HSync* program are defined as asynchronous parallel composition of the n copies of the algorithm in Listing 1. The code has the following structure: declaration of the variables that keep track of the timeout, the round number, and the buffer of delivered messages (lines 2-6), a few auxiliary methods (lines 9-19), and a big while loop where every iteration corresponds to one round (lines 22-49). All process goes thought all rounds in order. The while loop is further split into initialization (lines 24-26), send (line 28), message accumulator (lines 30-43), and finishing the round (lines 45-48). The message accumulator is a loop where at each iteration one message is received and processed. The termination of these loop triggers a round switch.

To ensure that eventually processes may communicate, the runtime (i) implements a catch-up mechanism that ensures the resynchronization and (ii) uses timeouts that defines how long a process waits for a message. For closed rounds, the message accumulator runs until a reference timeout is reached, whose value ensures resynchronization. For open rounds, if the methods `receive_init` and `receive` are implemented using `Progress.strictWaitMessages` then the runtime sets the timeout to be infinite.

The runtime uses the array phase consisting of *len* rounds defined in *HSync* and the variable `currentRound` which is the number of the round the process is currently in. In Fig. 1,

round[currentRound] is kth EventRound object defined in the *HSync* program, that is phase[k], where k is currentRound%len. For every round, the runtime uses the implementations of send, receive_init, receive, finishRound given in *HSync*, calling round[currentRound].send, etc. Each call is wrapped by checkProgress that implements the semantics of the progress conditions. For instance, checkProgress(*round*[*currentRound*].receive)) calls receive and then updates the timeout.

Reception is done using the method receiveWithTimeout in line 32 that either returns a message or null if the timeout (given by the *HSync* program) expired. For blocking algorithms like two-phase-commit receiveWithTimeout is called with an infinite timeout. In asynchronous executions, the messages delivered to a process contains messages from different rounds. For instance, if two processes p and q are in different rounds, and send messages for their current rounds to some process r, its message buffer will contain messages from different rounds, whose round numbers are not necessarily ordered. To address this, runtime implements a filtering of these messages. To each sent message it adds as metadata the current round number of its sender. When a message is received, the runtime discards it, if it was sent in a round smaller than the current round of the receiver, and it buffers it in the pendingMessages buffer if it is a message that comes from a future round. The mailbox contains messages only from the process's current round. The default behavior of receive_init is to set a timeout and the default behavior of receive is to add the received message to the mailbox, when it belongs to the current round. Messages in pendingMessages are moved to the mailbox when the process reaches the round they were sent in.

To ensure resynchronization after bad periods, runtime executions may *jump over* over some rounds so that a process can *catch-up* with processes that made quicker progress. Catch-up is a mechanism used by many asynchronous systems, e.g, Paxos, ViewStamped, PBFT. To enable this catch-up optimization, the strict flag must be set to false in the configuration of the runtime. The idea is that when a message m from a future round *fr* is received, instead of storing it and continuing with the current round, the runtime jumps to the round *fr* and starts the message accumulator of *fr* with m in the mailbox. The implementation of catch-up uses two variables nextRound ranging over round numbers and strict a boolean, the flag from the configuration. In case of a jump, the message accumulator of the current round stops, and finishRound is executed for the current round. Then, the runtime will execute only the method finishRound of all rounds up to nextRound, because finishRound is the only round operation that has side effects.

Byzantine HSync programs. Executing byzantine protocols poses new challenges. Let us first consider the catch-up mechanism in the byzantine case. In the benign case, a process makes a catch-up upon a single message reception. With this mechanism, a Byzantine process that sends arbitrary round numbers may permanently desynchronize the system. To implement the catch-up mechanism in the byzantine case, the runtime keeps not only the buffer of pending messages but also an array maxRound that for each process stores the highest round value it sent in a message. Instead of jumping to the maximal round number, the runtime ignores the f highest round values (line 19) in the array maxRound, where f is the maximal number of Byzantine processes, and jumps to the f + 1st highest one (at least one correct process made progress until that round).

To ensure that this catch-up is efficient, the synchronization algorithm of the runtime ensures that $f+1$ correct processes are always synchronized. This group ensures that all correct processes catch-up to them quickly in a good period. This algorithm can be enabled by using a ByzantineRound from the *HSync* language.

Internally, the execution platform transforms a byzantine round into an EventRound with a specific message accumulator, that ensures for byzantine closed rounds, that $f+1$ correct processes stay within a bounded round distance. The main idea follows the clock-synchronization algorithm

from [Dwork et al. 1988], where all processes communicate with each-other, and where a process switches rounds only if it has received $2f + 1$ messages for that round, and two timeouts to ensure that between two round switches the waiting time is sufficiently large to ensure that every correct process can communicate with every correct process. Finally, the runtime uses an auxiliary method `processReceive` that takes as input the message received from the network, and checks that only one message per process per round is delivered. This deals with malicious processes which may send multiple messages and faults in the transport layer, e.g., UDP may duplicate packets.

Listing 1. *HSync* Runtime Algorithm

```
// state of the runtime
timeout, roundStart, currentRound, nextRound = 0 // time and round numbers
timedout = false
maxRound: Array[ℕ] = ... // keep the max round seen for each process (used for deciding when to catch-up)
pendingMessages = ... //buffered messages (reordering and asynchrony)
mbox = []

// auxiliary methods
def processReceive(message) {
  if (∀ m ∈ mbox. m.sender ≠ message.sender) { // check duplication (UDP or Byzantine)
    mbox = message :: mbox
    checkProgress(round[currentRound].receive(message.sender, message.payload))
} }
def checkProgress(p: Progress) {
  if (p.isTimeout) { timeout = p.timeout; strict = p.isStrict }
  else if (p.isGoAhead) { strict = false; nextRound = max(nextRound, currentRound + 1) }
  else if (p.isWaitMessage) { timeout = ∞; strict = p.isStrict }
}
def catchUpTo() = maxRound.sorted[n - f - 1] //highest round excluding Byzantine

// each iteration of the loop corresponds to one round
while(true) {
  // RECEIVE INIT
  strict = false; timedout = false; mbox = [] // clean
  roundStart = currentTime() // set time
  checkProgress(round[currentRound].init())
  // SEND
  checkProgress(round[currentRound].send()) // check progress as sending to self can trigger a receive
  // RECEIVE
  for ( message <- pendingMessages.get(currentRound) ) processReceive(message) // deliver buffered messages
  while (nextRound == currentRound ∨ strict ) { // has not yet received enough messages
    message = network.receiveWithTimeout(roundStart + timeout - currentTime())
    if (message == null) { // timeout
      timedout = true; strict = false
      nextRound = max(nextRound, currentRound + 1)
    } else {
      maxRound[message.sender] = max(maxRound[message.sender], message.round)
      if (message.round < currentRound) { // late message, ignore
      } else if (message.round == currentRound) { (checkProgress(message))
      } else {
        pendingMessages.add(message) // buffer and check if we need to catch-up
        nextRound = max(nextRound, catchUpTo)
  } } }
  // FINISHROUND
  round[currentRound].finishRound(mbox)
  currentRound += 1
  maxRound[id] = currentRound
  nextRound = max(nextRound, currentRound)
}
```

Runtime executions. checkProgress(*round*[*currentRound*].send), checkProgress (*round*[*currentRound*].init), etc. can be considered to be executed atomically: The result of their execution depends only on the input parameters. During their execution there are no interaction with other processes (or the delivery buffers). Timeouts control only whether the methods are called, but do not influence their termination (once called). We refer these actions by *send*, *init_receive*, *receive*, *finishRound*, respectively. To formalize the semantics of the runtime, we consider also three additional atomic actions *discard*, *store* and *to*: *discard* is the action of receiving a message in line 32, followed by taking the branch in line 38 that discards the message, and *store* is the action of receiving followed by a call to pendingAdd in line 41 storing messages belonging to higher rounds, and *to* (timeout) decreases the time interval the process spends in the current round.

Definition 5.1 (Runtime executions). Given a *HSync* program $\mathcal{P}$, the set runtime executions, denoted $[\![P]\!]_{Run}$, is defined by the asynchronous parallel composition of n transitions systems, where each transition system defines the semantics of the code in Fig. 1. For *HSync* programs that use byzantine rounds, we consider that the runtime executes the inlined definition of ByzantineRound into EventRound. Formally,

$$[\![P]\!]_{Run} = ||_{p \in P} \left((\sigma_1 \vee \sigma_1^{jump}); (\sigma_2 \vee \sigma_2^{jump}); \dots \right)$$

$$\sigma_i = \; send_{i,p}; \left(\begin{array}{c} init\_recv_{i,p}; (to; (discard \vee recv^{j}_{i,p} \vee store \vee))^*; to\_stop; \\ \vee \\ init\_recv_{i,p}; (discard \vee recv^{j}_{i,p} \vee store)^*; \end{array} \right) finishRound_{i,p}; i{+}{+}$$

$$\sigma_i^{jump} = (finishRound_{i,p}; i{+}{+}) \vee i{+}{+}$$

where σ_i is the code executed by a process in round i where all methods are executed, σ_i^{jump} is the code executed at runtime in case of round jumps, and || is the asynchronous parallel composition applied on the sequence of actions executed by each process.

5.3 Runtime correctness

The execution platform preserves the properties of the *HSync* system that are closed under indistinguishability. Indistinguishability is an equivalence relation over executions, such that two executions are equivalent if a process goes through the same sequence of states in both executions, modulo stuttering. These properties are also called local properties in [Chaouch-Saad et al. 2009], and important class of specifications for fault-tolerant systems, such as consensus, k-set agreement, leader election, are local or closed under indistinguishability. We will later also highlight that for specific clients these results imply observational refinement.

Definition 5.2 (Indistinguishability). Given two executions π and π' of a transition system *TS*, a process p cannot distinguish locally between π and π', denoted $\pi \simeq_p \pi'$, iff the projection of both executions on p agree up to finite stuttering.

Two executions π and π' are *indistinguishable*, denoted $\pi \simeq \pi'$, iff no process can distinguish between them, i.e., $\forall p \in P.\ \pi \simeq_p \pi'$. Two executions π and π' are *indistinguishable w.r.t. a set of actions* A, denoted $\pi \simeq_A \pi'$, iff for every process p. the projection of both executions on p and on the actions in A agree up to finite stuttering.

Our first reduction theorem is concerned with the case where the designer implements the message accumulator, that is, for open rounds. The runtime ensures indistinguishable executions. However, we cannot ensure any message predicate different from *true* for open rounds.

Theorem 5.3 (Correctness – Open). *Given an open HSync program $\mathcal{P}[true]$, for every execution $ae \in [\![(\mathcal{P})]\!]_{Run}$, there exists an indistinguishable execution $se \in [\![\mathcal{P}]\!]$ with respect to the actions $C = \{init, send, init\_recv, recv, finishround\}$ under the two semantics, that is $ae \sim_C se$.*

Proof sketch. Let us consider an asynchronous execution $ae \in [\![(\mathcal{P})]\!]_{Run}$. The main ingredient is that the runtime only provides messages for the current round to the *HSync* program. Thus, reduction arguments [Chaouch-Saad et al. 2009; Damian et al. 2019; Elrad and Francez 1982] due to communication closure allows us to reduce to indistinguishable executions where globally actions are ordered with non-decreasing round numbers. Then we use the fact that send and receive are left and right movers [Lipton 1975], resp. This generates indistinguishable executions that have all the *init* and *send* as well as the $finishround$ for the same round appearing in a block, so that they can be reduced to synchronized actions as required for $se \in [\![\mathcal{P}]\!]$.

Timeout actions *to* (and additional sends for byzantine rounds) in the asynchronous executions just lead to stuttering invisible events, which we may drop and maintain indistinguishability. □

In the case of closed rounds our runtime controls message reception and round switch. This in addition to the reduction result, we can guarantee the message predicate *PS*. In the benign case the following theorem follows the same arguments as [Drăgoi et al. 2016] as our algorithm coincides with theirs, in the Byzantine case, our algorithm and correctness arguments follow the ones from [Dwork et al. 1988].

Theorem 5.4 (PS Correctness – Closed). *Given a Sync-HSync program $\mathcal{P}[PS]$, for every execution $ae \in [\![(\mathcal{P}[PS])]\!]_{Run}$ there exists an indistinguishable execution $se \in [\![\mathcal{P}[PS]]\!]$ with respect to the common events $C = \{send, init\_recv, recv, finishround\}$ under the two semantics, that is $ae \sim_C se$, where PS is the partial synchrony predicate defined in Sec. 4.3.*

For specific clients, the above theorems imply results regarding observational refinement:

Definition 5.5 (Observational refinement). Given two transition systems $TS1$ and $TS2$ and a set of actions of the two systems O, called *observable actions*, $TS1$ observationally refines $TS2$, denoted $TS1 \triangleright_O TS2$, iff all executions of $TS1$ projected on the observable actions O are included in the executions of $TS2$ projected on the observable actions O.

The only observable actions of a *HSync* program are the ones in the interface, e.g., `in()` that gets a value from a client and `out()` that returns a value to a client. Given that the runtime and the *HSync* semantics are indistinguishable w.r.t. `init` and `finishRound`, indistinguishable for interface actions follows. In [Filipovic et al. 2009] indistinguishability is proven equivalent with observational refinement for commutative clients, i.e., when the interface operations commute. The order between the input and the corresponding output is preserved by the program order, and from the client's perspective multiple equal outputs for the same input commute. We obtain:

Corollary 5.6. *Given a program $\mathcal{P}$ the runtime semantics of $\mathcal{P}$ observationally refines the HSync semantics of $\mathcal{P}$ w.r.t. the actions O in the interface of $\mathcal{P}$, if the interface actions commute for the client, that is, $[\![(\mathcal{P}[true])]\!]_{Run} \triangleright_O [\![\mathcal{P}[true]]\!]$ and $[\![(\mathcal{P}[PS])]\!]_{Run} \triangleright_O [\![\mathcal{P}[PS]]\!]$.*

5.4 Design Principles for Open Rounds

If the designer implements the message accumulator, and thus uses open rounds, the runtime does not control waiting and consequently cannot ensure the useful message predicate *PS*. We review common communication patterns and network models and communication patterns in distributed algorithms in order to show what the designers have to ensure so that their implementation ensures *PS*, that is, from some round on, all messages between correct processes are delivered.

Simple system model. In this section, we look at a model inspired by the partial synchrony model [Dwork et al. 1988]. At the beginning, the system may be arbitrarily desynchronized as result of having no bound on message delay or relative speed of the processes (scheduling fairness). The processes always send messages according to a predefined pattern. During the good period, we know the message transmission delay between any two processes. We write $\Delta(p, q)$ for the time it takes for a message sent by p to be received at q. We further assume that processing messages does not take time and no crash occurs during a good period.

Let $s(r, p)$ be the time at which process p enters the round r. Let $o(r, p, q) = s(r, q) - s(r, p)$ be the time difference distance between p and q at the beginning of round r. $o(r, p, q)$ is negative if q is ahead of p and we have that $o(r, p, q) = -o(r, q, p)$.

Example 5.7. Let us consider example of violating *PS*, even during good period. Consider a system with three processes A, B, and C. In each round, every process sends a message to all. Processes waits for messages and go ahead as soon as they have received all the messages for a round. A process tries to catch-up with the first message with an higher round number. Regarding the network, the message transmission delay is 1 time unit between each process pair except for messages from A to C where $\Delta(A, C) = 3$. The system starts synchronized and is always in a good period. For the first round we have $s(0, A) = s(0, B) = S(0, C) = 0$. At time unit 1, A and B receive 3 messages and C receives 2 messages. A and B moves to the second round and C waits for one more message. A and B both sends their respective messages for the second round. At time unit 2, C receive the B's message for the second round. Therefore, it catches up and will discard A's message for the 1st round when it arrives at time unit 3, so that *PS* is violated.

We now cover idiomatic communication patterns in distributed algorithms, along with proof strategies one can use to prove that a specific program progresses with message predicate *PS*.

5.4.1 All-to-all communication. This is the case of Example 5.7. Instead of modifying the algorithms, we make the extra assumption on the network, namely that the network delays respects triangle network inequalities. More formally, we require that $\Delta(p, q) \leq \Delta(p, o) + \Delta(o, q)$ for any triple of processes p, q, and o. Intuitively, the triangle inequalities prevent reordering where a message from the next round overtake a message from a past round. To show that the system ensures *PS*, the designer can prove the formula $\exists r_0.\ \forall r, p, q.\ r \geq r_0 \Rightarrow o(r, p, q) \leq \Delta(p, q)$, which states that from some round r on, any two processes start the round within a message delay. One arguments that r_0 is the first round which is started after the good period started, and uses r_0 as the base case for an induction of the round numbers. The catch-up will ensure that all processes enter round r_0 within a message delay. One then continues the induction from the assumption r_0 satisfies the property.[1]

5.4.2 All-to-one and one-to-all communication. While algorithms broadcasting to everybody are conceptually simple, most practical algorithms use a leader with which every other processes communicate. With a leader, only n messages are sent per round instead of n^2. This communication pattern is the one used by Paxos. Let us start with the same assumption as the previous scenario and have the program behaves as follows. c is a special coordinator process. During even rounds the coordinator sends to the other processes and, during odd rounds, the coordinator receives from the other processes. The processes also have a large enough timeout ($> 2max_{p,q}(\Delta(p, q))$. Similar to the previous scenario, the designer needs to prove:

$$\begin{aligned}\exists r_0.\ \forall r, p.\ \ & (r \geq r_0 \wedge r \equiv 0 \mod 2 \Rightarrow o(r, c, p) \leq \Delta(c, p)) \ \wedge \\ & (r \geq r_0 \wedge r \equiv 1 \mod 2 \Rightarrow o(r, p, c) \leq 0 \wedge o(r, c, p) \leq \Delta(c, p))\end{aligned}$$

[1]The proofs of this section, and more patterns can be found in the supplementary material .

Intuitively, the 1st rounds when the coordinator sends after GTS brings the other processes at most $\Delta(c, \_)$ behind the coordinator. For the following rounds, the coordinator keeps the system synchronized. Because the coordinator waits for the messages from all the processes, it progresses after all the other processes are already at the next round ready to receive.

6 IMPLEMENTATION AND EVALUATION

We have implemented *HSync* in Scala on top of the PSync [Drăgoi et al. 2016] codebase and evaluated it experimentally. The evaluation has two goals. First, we show that the *HSync* programming model is competitive against other implementations of distributed algorithms. Second, we study the effect of different progress conditions on the consensus algorithms.

6.1 Runtime Implementation Details

The points discussed below do not change the functionality of the runtime but improve efficiency. In particular, we discuss aspects related to memory management, which are not explicitly visible in the algorithm in Section 5 as its presentation focuses on the computation.

Typed rounds and serialization. From the user's perspective, the communication is typed, i.e. the rounds specify the type of the message payload. All the serialization is encapsulated within the rounds. The runtime itself is agnostic to the message content, it just moves bytes.

Since we consider Byzantine processes, serialization is a weak point and can often be abused by malicious processes. While hardening against that type of attack is outside the scope, we take basic protection measure. We use Kryo (https://github.com/EsotericSoftware/kryo) for the serialization which requires explicit registration of the types which can be deserialized. While we provide safe serializers for simple types, e.g, collections of primitive types, any serializer for more complex objects need to be provided by the user and properly hardened.

Self messages bypass. For clarity, distributed algorithms often contain instructions such as "send to all" or "wait for replies from a quorum". In both these cases, the process doing the action is included in the action's target. Sending a message to self would be wasteful. *HSync* detects these messages and handles them separately. In particular, these messages are kept within the round itself and directly forwarded to the `receive` method. This makes it possible to keep the program simple, i.e., treat the current processes just as another process, without incurring any extra cost.

Memory management. In Listing 1, a message with an higher round number first get stored in the `pendingMessages` buffer (line 41) and then directly taken out of the buffer for delivery (line 30). While this may looks harmless, this part has a non-negligible cost when executed very frequently. The cost is due to updating the `pendingMessages` data structure which allocates and frees memory. Therefore, for catch-up, the runtime avoids storing messages in the `pendingMessages` buffer.

Other memory management aspects are the cost of memory allocation and bounding the maximal amount of memory used by the runtime. Bounding the memory is essential when considering malicious attackers. Since we store messages that are supposed to be delivered later (`pendingMessages`), a malicious process could send many messages with very high round number and thus make `pendingMessages` grow arbitrarily large. Therefore, we put a bound on the maximal number of incoming messages per process that are stored. When there are too many messages, we evict the ones with the smallest round number.

The runtime is a fairly complicated object in itself and we do not want to recreate it each time we run a different algorithm which may just send and receive a few messages. The runtime and the rounds can be quite expensive to allocate as they contain other objects which also needs to

be allocated, e.g. `maxRound`, `pendingMessages`. Internally, we pool and reuse the objects with a complex internal structure as well as the memory buffers for sending and receiving messages.

Transport layer. The *HSync* model is agnostic to the transport layer. We support TCP and UDP in the benign case. Authenticate communication for Byzantine protocols is implemented using TLS on top of TCP[2]. This also protects from a large class of Sybil and replay attacks. Being able to precisely identify the sender and only receive from expected sender is necessary for the integrity of the model. The constant `n` only exists if we know the set of processes executing a program.

Round number and overflow. During an execution the round number can grow arbitrarily. While we could use arbitrary precision number to account for that grow, we decided to implement numbers with finite precision and take advantage of the wrap around semantics of integers on the JVM. This means a comparison $a \geq b$ becomes $a - b \geq 0$. The advantage of this solution is its simplicity and efficiency. Theoretically speaking, it is only correct as long as the distance between the fastest and slowest correct processes is smaller than half of the range of an `int` ($2^{31} - 1$).

6.2 Experimental Evaluation

In this section, we compare the implementation of the consensus algorithm in *HSync* against other implementation of the similar algorithms and we quantify the effect of progress conditions. We run our experiments on servers with 2 Intel Xeon X5650 (6 cores at 2.67 GHz), 48GB of RAM, and a gigabit network interface. The average ping between any two machines is around $0.17ms$. The servers run Debian stretch with Linux kernel 4.14 and we use the OpenJDK 11. As the JVM has a slow startup and *HSync* runtime allocates most resources at the beginning, to amortize the startup cost we report averages over 5 minutes runs.

6.2.1 Throughput. The round based model of *HSync* and the runtime that comes with it have a cost. We perform two experiments comparing consensus algorithms in *HSync* against existing implementations to evaluate that cost. We look at the case of benign and Byzantine faults.

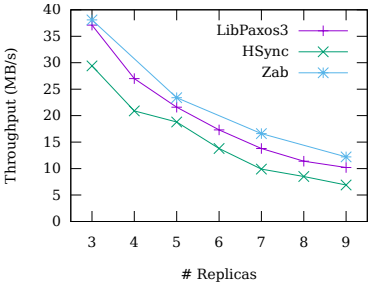
Benign consensus. First, we compare an implementation of the Paxos consensus algorithms in *HSync* against and implementation of the same algorithms in C. More precisely, we compare iterated version of Paxos (multi-Paxos) in *HSync*. It is an adaptation to open rounds of the round-based version of Paxos given in [Charron-Bost and Schiper 2009], called Last Voting. The algorithms is used in a key-value store. Each decision of the consensus algorithms handle a batch of requests. The batches contain 32KB worth of data. We measure the amount of data that the system can process per second. The results are shown in Fig. 11a. The throughput of the system is measured in MB per second. To test the scalability of the system, we run the test on 3 to 9 replicas.

We compared the *HSync* implementation of consensus with LibPaxos3 [Sciascia 2016], a C implementation, which has been used in the distributed algorithms community as baseline for comparison [Jha et al. 2019; Marandi et al. 2010]. The system processes requests which are simple arrays of bytes. We use requests size of 32KB. LibPaxos3 is faster by 20% to 35%. The difference becoming larger as the number of replicas increases. We also include the performance number from Zab [Junqueira et al. 2011] as reference [3].[4] The *HSync* implementation and LibPaxos3 follow the

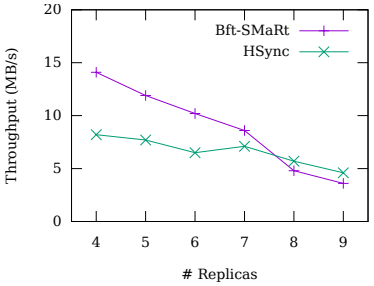
[2]The user of *HSync* still needs to provide the mechanism used to authenticate processes, e.g. check certificates.

[3] Benchmarking fairly distributed algorithm is difficult. Each system has a number of parameters which have to be tuned to achieve the best performances. Therefore, we want to include a comparison against a system tuned by its authors. The machines on which we run our experiments are rather old. We choose them as they match the evaluation of Zab performed in 2011 [Junqueira et al. 2011]. The system has very likely been evaluated on Intel Xeon X5630. The X5630 processors compared to the X5650 have 4 cores instead of 6 and run at 2.53 GHz instead of 2.67 GHz.

[4]In the supplementary material , we look in more details at the effect of batching of the system's throughput.



(a) Benign test: *HSync* against LibPaxos3 and Zab



(b) Byzantine test: *HSync* against Bft-SMaRt

Fig. 11. Comparison for benign and Byzantine Consensus Algorithm

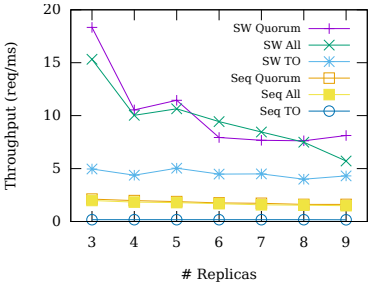
behavior exhibited by Zab which is used in industry. While there is a cost to the round abstraction it is not prohibitive. We hope to further reduce this gap by optimizing the runtime implementation.

Byzantine consensus. The second comparison is for a Byzantine case. We compare against Bft-SMaRt [Bessani et al. 2014] a java library for Byzantine fault-tolerant replicated state machines. We have implemented in *HSync* the normal decision algorithms from PBFT [Castro and Liskov 2002]. The algorithms uses a leader which sends requests to all the replicas. Then all the replicas perform two rounds of all-to-all communication to establish and confirm a quorum larger than $2n/3$ around that request. During the all-to-all rounds, only digests instead of the full requests are forwarded. We use SHA-256 to compute the digest of the request. The results are shown in Fig. 11b. We can observe that *HSync* performs worse when a small number of replicas is used, but the difference gets smaller when the number of replicas increases. We hypothesize that Bft-SMaRt uses more expensive cryptography and becomes CPU bound. On the other hand, *HSync* has a larger synchronization overhead due to Byzantine rounds which is more visible with a small number of processes. An interesting point to observe is the transition between 6 and 7 replicas for *HSync*. 6 replicas tolerate 1 fault and 7 replicas tolerate 2 malicious processes. In both case, 5 messages are expected for a quorum. Therefore, if some replicas are slow, e.g., due to garbage collection, it has less impact as *HSync* waits for the first 5 messages.

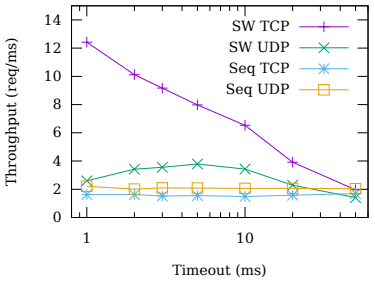
6.2.2 Comparison between different message accumulators. We performed more tests on the same algorithms but with different message accumulators. We perform most of our test on the LastVoting algorithm.[5] In this experiments, rather than maximizing the throughput using large requests which saturate the network, we try to generate many small packets which increase the load on the *HSync* runtime. We report the speed of the system by the number of requests where each request is processed independently. We use two different conditions of deployment:

- *Sequential* (Seq): In this condition, the system calling the consensus algorithm makes decisions sequentially. Only when one decision has terminated does the next decision start.
- *Sliding Window* (SW): In this condition, we use a sliding window to increase the system load. While a decision is still pending, the system already starts the next decisions. The size of the sliding window is the limit on how many decision can be performed in parallel. The decisions can happen out-of-order but they are reordered before being applied to the system. We use a sliding window of size 20. This sliding window is a common optimization and, for instance, LibPaxos3 also uses it.

[5]In the supplementary material , we show that similar results can be observed on the two phase commit algorithm.



(a) Comparing progress conditions with TCP transport and a 5ms timeout



(b) Effect of timeout values and transport layer with 9 replicas progressing on quorum

Fig. 12. Comparing different progress conditions with the Paxos consensus algorithm

Waiting on some messages, all messages, or the timeout. We modified our Paxos implementation to test how different waiting conditions affect the algorithm. The replicas progress as soon as they receive message from the leader. On the other hand, the leader either progress when it received messages from a quorum or timeout (Quorum), progress when it received messages from all the replicas or timeout (All), or timeout (TO). The results are shown in Fig. 12a.

We see that the (TO) approach is much slower than the rest. For (Quorum) and (All), we can observe some interesting behaviors. The throughput is not monotonically decreasing with the number of replicas. These behaviors can be explained by the waiting condition of the consensus algorithm: To progress, the leader needs to receive messages from a quorum of processes which includes the leader itself. So for 3 replicas, the 1st out of 2 messages leads to progress. For 4 replicas, the leaders needs to receive 2 out of 3 messages and, for 5 replicas, 2 out of 4 messages. So each time we transition from an odd to even number of replicas the leader waits for one more message. When going from an even to odd number of replicas, the leader waits for the same number of messages but there is one more process sending. This effect is particularly pronounced when looking at the (Quorum) tests and, in a lesser extend, is also visible in the (All) tests.

Varying the timeout. One critical parameter is finding an appropriate timeout value to detect crashes. Compared to other approaches where timeout is only supposed to detect crashes, it may be better to *not* be conservative when picking the timeout in *HSync*. *HSync*'s round synchronization happens with messages and therefore, timing out early and retrying results in sending more messages to synchronize the system. Fig. 12b show what happens when running Paxos on 9 replicas for different timeout values over either TCP or UDP. For this test, we use timeout values of 1, 2, 3, 5, 10, 20, and 50ms. Recall that the network latency is 0.17ms and *HSync* timeout granularity is 1ms. It is interesting to observe that in the Seq condition, the system is sightly faster over UDP than TCP. On the other hand, with an higher load TCP has a clear advantage. With higher load, we can observe that the performance starts degrading when the timeout get below 5ms with UDP while TCP makes a much better jobs at keeping the system synchronized with a small timeout.

7 RELATED WORK AND CONCLUSION

General purpose programming languages are largely used to implement distributed systems. For instance, C/C++ (Redis [Sanfilippo 2015], a popular key/value store), Java (Zab, Zookeeper's atomic broadcast protocol [Junqueira et al. 2011], Cassandra distributed database system, Bft-SMaRt [Bessani et al. 2014], a Byzantine fault-tolerant state machine replication), Haskell (Raft [Ongaro and Ousterhout 2014]), Ocaml (Verdi [Wilcox et al. 2015]). Programmers chose them for their flexibility, for the prize of vulnerability to bugs. Therefore these implementations are often subject

to critical conceptual bugs [CASSANDRA 2013; Jepsen 2018], like violations of the specification of consensus, or intolerance to faults.

Among the programming languages for distributed systems are Go [Griesemer et al. 2009] and Erlang[Armstrong et al. 2019] which are among the most popular ones, or more recently Rust [Rust 2019]. Go has communicating sequential processes (CSP), and offers memory safety and a garbage collector. Notable examples of software implemented in Go are part of Google Cloud, Amazon Web Services, Microsoft Azure. Erlang is a functional programming language where everything is a processes, with pattern matching, and implicit message reception based on timeouts. Notable projects implemented in Erlang are Amazon SimpleDB, Antidote, Riak.

Programming languages for distributed systems have been developed also with the goal of formal guarantees. Important work are the language P [Desai et al. 2013], for asynchronous event-driven applications. It has a dedicated specification language and a testing tool for concurrency bugs. Ivy [Padon et al. 2016] is framework that uses a high-level language to specify and implement systems, and do verification. TLA+ [Lamport 2002] is a specification language, that captures a large class of systems and comes with an explicit model checker and a theorem prover, but it is not executable. Languages like Go, Erlang, Rust, simplify programming for the distributed setting, while P and Ivy allow powerful testing and verification. However, they are all either rooted in general programming languages or too focused on verification, and lack synchronization primitives that are inherent to fault-tolerance, such as an abstract notion of logical time [Fidge 1991].

The systems mentioned above, e.g., Cassandra, EPaxos, Bft-SMaRt, etc., implement a logical notion of time, implementing for example vector clocks [Mattern 1989] or Lamport clocks [Lamport 1978]. Synchronous round based models [Lynch 1996] with their inherent logical time provided by the round number, relive the programmer from implementing any notion of logical time. Partially synchronous round-based models [Charron-Bost and Schiper 2009; Dwork et al. 1988; Gafni 1998] were the inspiration and the main motivation of this work. Our programming abstraction extends round-based models by allowing the programmer to implement custom optimizations for the round-switch, addressing one of the main concerns raised w.r.t. the performance of round structures. The closest related work is PSync [Drăgoi et al. 2016] that we build upon. PSync has a communication-closed round-based semantics similar to *HSync* that compiles to efficient asynchronous code, and has a verification engine based on deductive verification. Compared with *HSync*, PSync does not tolerate Byzantine faults, PSync may become desynchronized and never recover. PSync does not allow the algorithm designer to explore and implement optimizations for specific networks. Algorithms that run under strong network assumptions like two-phase-commit are not implementable in PSync (which does not allow algorithms to block until a certain condition is reached).

Similarly to round models, virtual synchrony [Birman and Joseph 1987] gives the illusion of synchrony on top of an asynchronous network. The class of system expressible in virtual synchrony is incomparable with *HSync*. Virtual synchrony does not allow custom optimizations algorithms, and also it does not have high-level concepts to facilitate the implementation of byzantine protocols.

The benefits of synchrony are highlighted not only from a design perspective but also from a verification perspective. Synchronous programs are easier to automatically verify, as shown in [Bouajjani et al. 2018; Damian et al. 2019; von Gleissenthall et al. 2019]. Our approach maintains communication closure therefore keeps simple correctness arguments, adapted to many verification frameworks [Aminof et al. 2018; Dragoi et al. 2014; Marić et al. 2017; Stoilkovska et al. 2019].

In conclusion *HSync* is a new programming abstractions that allows the implementation fault-tolerant distributed protocols in a modular way — separating message accumulation from round-based computation — and adapting round switching policies relative to the network the code should run on for. *HSync* is one infrastructure that works both in the benign and Byzantine case, and implements a default round switch logic that provides theoretical guarantees. We compared *HSync*

implementations of benign and byzantine implementations of consensus and atomic broadcast with reference implementations from the literature, like LibPaxos3 and Bft-SMaRt, and showed that the runtime of *HSync* is competitive in performance with these reference systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger. 2018. Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction. In *VMCAI*. 1–24.

Joe Armstrong, Robert Virding, Mike Williams, and Ericsson. 2019. Erlang. https://www.erlang.org

Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Dahlia Malkhi Zekun Li, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. State Machine Replication in the Libra Blockchain. https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf.

Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 355–362. https://doi.org/10.1109/DSN.2014.43

K. Birman and T. Joseph. 1987. Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Oper. Syst. Rev.* 21, 5 (Nov. 1987), 123–138.

Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. 2018. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *CAV*. 372–391.

Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Master's thesis. University of Guelph. http://hdl.handle.net/10214/9769.

CASSANDRA. 2013. Bug report. https://issues.apache.org/jira/browse/CASSANDRA-6023.

Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (1996), 225–267.

Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. 2009. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *RP (LNCS)*, Vol. 5797. 93–106.

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing* 22, 1 (2009), 49–71.

Ching-Tsun Chou and Eli Gafni. 1988. Understanding and Verifying Distributed Algorithms Using Stratified Decomposition. In *PODC*. 44–65.

Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. 2019. Communication-closed asynchronous protocols. In *CAV*. (to appear).

Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 321–332.

Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-Based Framework for Verifying Consensus Algorithms. In *VMCAI*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 161–181.

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*. 400–415.

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *JACM* 35, 2 (April 1988), 288–323.

Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982), 155–173.

Colin Fidge. 1991. Logical Time in Distributed Computing Systems. *Computer* 24, 8 (Aug. 1991), 28–33.

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer, 252–266. https://doi.org/10.1007/978-3-642-00590-9_

19

Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. 1985. Impossibility of Distributed Consensus with one Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

Eli Gafni. 1998. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony (Extended Abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, Brian A. Coan and Yehuda Afek (Eds.). 143–152.

Robert Griesemer, Rob Pike, and Ken Thompson. 2009. Go. https://www.erlang.org

Jepsen. 2018. Distributed Systems Safety Research. Web page jepsen.io. Retrieved Nov 7, 2018.

Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2 (2019), 4:1–4:49. https://doi.org/10.1145/3302258

Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *DSN*. 245–256.

Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4 (2009), 7:1–7:39.

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

Leslie Lamport. 2002. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley.

Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report. 60 pages. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Nancy Lynch. 1996. *Distributed Algorithms*. Morgan Kaufman.

Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. 2010. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*. IEEE Computer Society, 527–536. https://doi.org/10.1109/DSN.2010.5544272

Ognjen Marić, Christoph Sprenger, and David A. Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *CAV*. 217–237.

Friedemann Mattern. 1989. On the relativistic structure of logical time in distributed systems. *Parallel and Distributed Algorithms* (1989), 215–226.

Brian M. Oki and Barbara Liskov. 1988. Viewstamped Replication: A General Primary Copy. In *PODC*. 8–17.

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 305–319.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. 614–630.

Rust accessed July, 2019. Rust (Web page). https://www.rust-lang.org.

Salvatore Sanfilippo. 2015. REDIS. http://redis.io/.

Daniele Sciascia. 2016. LibPaxos3. Retrieved July 9, 2019 from https://bitbucket.org/sciascid/libpaxos/

Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. 2019. Verifying Safety of Synchronous Fault-Tolerant Algorithms Bounded Model Checking. In *TACAS, Part II (LNCS)*, Vol. 11428. 357–374.

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL* 3, POPL (2019), 59:1–59:30. https://doi.org/10.1145/3290372

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 357–368.