# Programming at the edge of synchrony

Cezara Dragoi, Josef Widder, Damien Zufferey

# Programming at the Edge of Synchrony

CEZARA DRĂGOI, INRIA, ENS, CNRS, PSL*, France
JOSEF WIDDER, Informal Systems, Austria
DAMIEN ZUFFEREY, MPI SWS, Germany

Synchronization primitives for fault-tolerant distributed systems that ensure an effective and efficient cooperation among processes are an important challenge in the programming languages community. We present a new programming abstraction, ReSync, for implementing benign and Byzantine fault-tolerant protocols. ReSync has a new round structure that offers a simple abstraction for group communication, like it is customary in synchronous systems, but also allows messages to be received one by one, like in the asynchronous systems. This extension allows implementing network and algorithm-specific policies for the message reception, which is not possible in classic round models.

The execution of ReSync programs is based on a new generic round switch protocol that generalizes the famous theoretical result of ?. We evaluate experimentally the performance of ReSync's execution platform, by comparing consensus implementations in ReSync with LibPaxos3, etcd, and Bft-SMaRt, three consensus libraries tolerant to benign, resp. byzantine faults.

CCS Concepts: • **Software and its engineering → Domain specific languages**; *Control structures*; *Runtime environments*; • **Theory of computation → Distributed computing models**;

Additional Key Words and Phrases: Distributed systems, Fault-tolerance, Round Model, Synchrony

**ACM Reference Format:**
Cezara Drăgoi, Josef Widder, and Damien Zufferey. 2020. Programming at the Edge of Synchrony. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2020), 31 pages.

## 1 INTRODUCTION

Fault-tolerant distributed algorithms are notorious for being hard to design and implement. One needs to take into account features of a non-deterministic network, behaviors of faulty processes, as well as the asynchrony of the actions performed by correct processes. These algorithms underpin many distributed applications and critical services, e.g., Zab [?], Zyzzyva [?] or more recent blockchains like Tendermint [?] or Libra [?]. These systems are implemented in general-purpose programming languages that support the asynchronous programming paradigm. This programming paradigm is error-prone and facilitates bug occurrences [???] even when implementing protocols that have been formalized and proved like Paxos [?]. These bugs show that going from (verified) protocols to implementations is a difficult task. The asynchronous programming paradigm, despite its pitfalls, is preferred because it enables important performance optimizations but also because other suitable programming abstractions and synchronization primitives are lacking. Therefore, our goal is to identify new synchronization primitives that simplify programming distributed systems, tolerant to benign and byzantine faults.

The seminal work [?] defines the *basic round* model, the first abstract computational model that offers rounds as a synchronous primitive. The model establishes its notoriety because it can solve consensus, i.e., $n$ processes trying to agree on a value, provided that the network satisfies the *partial synchrony* assumption. Consensus is not solvable over asynchronous networks in the presence of faults [?] and *partial synchrony* is one of the weakest theoretical assumptions that allows us to

Authors' addresses: Cezara Drăgoi, INRIA, ENS, CNRS, PSL*, France, cezara.dragoi@ens.fr; Josef Widder, Informal Systems, Austria, josef@informal.systems; Damien Zufferey, MPI SWS, Germany, zufferey@mpi-sws.org.

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

solve it. Partial synchrony means that the network eventually becomes synchronous, i.e., there exists a bound Δ on the message delay and a bound Φ on the relative speed of processes, however these bounds hold only from some time on, called global stabilization time GST. (There are two other definitions of partial synchrony in [?], which are less used.) The basic round model, and the follow-up round-based approaches [???], rely on an abstract global clock. The abstract global clock synchronizes all processes, that is, all processes execute the code of a round in lockstep (at the same time), where a round is a sequence of sending messages, receiving (some or all) messages sent in that round in one atomic step, followed by some local updates based on the received messages. To "execute" a consensus-solving algorithm in a round model, one has to implement an algorithm that computes when the round switch should locally happen such that processes switch rounds at the same time. To this, ? use an algorithm that implements a distributed clock, exploiting Δ and Φ from the partially synchronous network assumption. The distributed clock determines the round switch at each process. Roughly, before GST the system is in an arbitrary state (different processes may have different clock values), but after GST, the distributed clock is guaranteed to simulate the abstract global clock and implicitly synchronizes all the correct processes. This is a purely theoretic result, since it requires multiple synchronization steps (clock steps) within a round, and it makes simplistic assumptions regarding the time spent by processes sending and receiving messages or performing local computation. In general, round-based solutions are infamous for their prohibitive performance, although most existing solutions for consensus rely on the partial synchrony assumptions and implicitly on rounds.

Among the systems that rely on partial synchrony are PBFT [?], BftSmart[?], ViewStamped Replication [?], Zookeeper's atomic broadcast protocol [?], Apache Cassandra [?], libPaxos [?], or the more recent works emerging from blockchain research like Tendermint [?], RedBelly [?], Lumière [?], Hot-Stuff [?], or Libra [?]. Each of these systems, use rounds implicitly and propose a new algorithm for the round switch that exploits partial synchrony in a different way, thus resulting in a new solution for consensus (state machine replication) either in the benign, or in the Byzantine case. Although solving the same problem under similar network assumptions, these are stand-alone systems, sharing no high-level control structure. They are implemented in error prone asynchronous programming paradigms (as shown by numerous bug reports), which are preferred for facilitating performance optimizations. These systems exploit the synchronicity of the network after GST differently, and they guarantee that soon after GST, correct processes are synchronized (a property called *eventual synchrony* [?]), and reach consensus. To implement eventual synchrony, these systems use asynchronous features like properties on the set of received messages (e.g., a message with a certain payload or from a certain sender has been received, or a certain number of messages have been received) that trigger a local round switch or timeouts on the local duration of a round, which ensure that after GST the processes get synchronized. These features are opaque in the basic round model as they are hidden in the theoretical implementation of the distributed clock. In the basic-round model, and in all follow-up round-based programming abstractions, processes have no control over the set of received messages and no control over the round switch.

*Problem statement.* In this paper we ask the following question: is there a programming abstraction that systematizes the optimization principles used in implementations of consensus protocols for partially synchronous networks? More widely, is there a synchronous round-based model that allows protocol specific optimizations, which today are possible only in asynchronous programming frameworks?

We propose ReSync, a new programming language with a novel round structure which preserves the abstraction level of theoretical round-based models and reasonably matches the performance demands of large-scale systems like Bft-SMaRt [?] or LibPaxos3 [?]. ReSync is a parametric framework that accommodates various implementations of the eventual synchrony property, rather than a fixed implementation, as it is the case in all previous works. We systematize the optimization principles used in large-scale systems that solve consensus and wrap them into a domain-specific language, that facilitates future consensus implementations under partial synchrony, and in general facilitates exploiting the network specifics.

ReSync is addressed to the algorithm designer who wants to prototype algorithms without worrying about network code and data management, and to the programmer less versatile in reasoning about interleavings and partial synchrony.

*Key insights.* In synchronous (or round-based) models, computations are structured in a sequence of rounds, where in each round all processes **send** messages and then **update** their local state based on the set of received messages. Rounds are communication-closed [?] and are a powerful synchronization primitive: messages are not received outside the round they were sent. The message reception of a round is hidden from the protocol's perspective: a non-deterministically chosen subset of the messages sent is received in one atomic step. Round structures are known for their poor performance due to an excessive need of synchronization. Any execution framework for the round-structure must implement a round switch algorithm. All known approaches are either too general [?] and hence purely theoretical due their poor performance, or they are restricted to a sub-class of distributed protocols like PSync [?] which implements one solution for eventual synchrony (that is not customizable). It is well established that there is no unique round switch algorithm that fits the performance needs of all protocols. Therefore, to design a more flexible way to implement the round switch we turn to asynchronous computational models.

Asynchronous computational models are preferred for implementing systems due to their better performance. Still, algorithm designers, present their asynchronous solutions in terms of rounds, phases, epochs, ballots, etc., e.g., [???]. 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 enables network-specific performance optimizations: messages are received one-by-one, which allows the system to react quickly to incoming messages, as opposed to the round structure that would wait for all the messages of the current round before reacting. For example, under the asynchronous semantics a process could switch rounds fast, as soon as it receives a message that contains the information it was waiting for, e.g., the first message that is receives comes from the leader and contains the value all processes agree on. In contrast, when executing the synchronous round-based model, a process would wait for all the messages sent in that round, even if the followers (the processes that are not leaders) only relay the message from the leader for fault-tolerance purposes.

The main insight is that we want *a round structure where messages are received one by one*, instead of all in one atomic step. Therefore, we define a new round structure where a round combines:

**Sending** messages,

**A message accumulator** that is a new round component which allows programming the message reception within the round-based model, and computes the sequence of messages received in a round, and

**A round computation** that **updates** the local state, depending on the mailbox of the current round computed by the message accumulator.

The executions of the new round model preserve the lockstep semantics at the round boundaries, but, within a round, messages are received asynchronously by the message accumulators of different processes. In this way, messages are received one-by-one and the accumulator decides when to wait for more messages or transition to the next round. This removes the need for extra synchronization, i.e., synchronization barriers needed to satisfy the (conservative) guarantees of the round model but which are not required for the protocol-specific progress condition.

The implementation of the message accumulator, more precisely the termination of a message accumulator, uses new features that are not available in any other round-based programming abstraction. First, the language provides constructs to control the termination of a message accumulator using timeouts, or using properties satisfied by the received messages for the current round. These features are sufficient to program message accumulators for distributed protocols whose specification is solvable in asynchronous networks.

However, timeouts and conditions on the set of messages received for the current round are not sufficient for implementing a round-switch in consensus protocols. The implementations of consensus exploit the partially synchronous nature of the network to synchronize processes after GST. For that, they use features that are not communication-closed, i.e., they cannot be implemented directly in a round-based model, e.g., a process reacts to messages sent in rounds different from the process's current round. The key to ensure synchronization is to fine-tune the moment when processes locally switch rounds, such that after GST all correct processes are roughly in the same round. The number of correct processes is determined by the system being tolerant to byzantine or benign faults. Byzantine systems are parametrized by the number of tolerated byzantine processes, denoted $F$. Instead of using a distributed clock to define the round switch, like in [?], we use hardware clocks and we introduce two synchronization primitives that trigger the termination of the message accumulator:

**Catch-up.** The message accumulator terminates and the process switches to a future round $r$, when it receives $F + 1$ messages sent in rounds greater than or equal to $r$, where $F$ is a parameter of the system, the maximal number of byzantine processes.

**Synchronize $F + 1$ processes.** The message accumulator terminates and the process switch to the next round only if it observes that at least $F + 1$ correct processes switched to the next round, where $F$ is again the number of byzantine processes.

Each round is parametrized by the enabling/disabling of these primitives for protocol-specific values of $F$. To compute a round switch, the runtime of ReSync integrates the implementation of these primitives (when enabled) with the timeouts (or other conditions) defined in the message accumulator. We show that the execution model implemented by the runtime of ReSync matches the theoretical guarantees for eventual synchrony provided in [?], under similar assumptions.

*Evaluation.* We evaluated ReSync on one benign and one byzantine implementation of Paxos and compared it against LibPaxos3 [?], an open source implementation of the Paxos algorithm, etcd an implementation of consensus based on Raft [?], and Bft-SMaRt [?], an open source implementation of byzantine agreement. ReSync performance are at most 30% worse than LibPaxos3 for a small number of replica, and 10% faster with more replicas. The results are show a 3.5× improvement of ReSync over PSync and ReSync and 25× faster than Goolong [?].

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

- We propose a new round structure that lets programmers use asynchronous optimizations without breaking the synchronous structure.
- ReSync is suitable for both benign and byzantine fault-tolerant protocols.
- ReSync allows the algorithm designer to write custom code that locally controls the round boundaries, i.e., the message accumulator with synchronization primitives, expressing a wide range of protocols.

- ReSync 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 ?.

## 2 OVERVIEW

A program in ReSync is parametric in $N$, the number of processes, and $F$, the number of byzantine ones among them. All (correct) processes execute the same code. A program is a sequence of rounds, called *phase*. In a round, processes send and receive messages, and then update their local state (depending on the set of received messages). ReSync has a round-based semantics, which facilitates writing programs, and an equivalent runtime asynchronous semantics to execute programs efficiently. In the following, we focus on the round-based semantics of ReSync, where rounds proceed in lockstep. Thus, there are no interleaving of actions performed by processes in different rounds.

*Example.* As example protocol, we consider a round-based version of ViewChange [?], a protocol used in PBFT [?]. The goal of PBFT is to receive requests from a client, and to store all of them on $N$ replicas, in the same order, tolerating $F$ byzantine faulty processes, where $N = 3F + 1$. Ideally, all replicas have the same history of client requests, however due to faults this is not the case. ViewChange ensures that a majority of correct replicas—i.e., at least $F + 1$ correct replicas—agree on the most recent history of requests. To this, it elects a leader, which makes sure that all the replicas in its quorum agree on the most recent history.

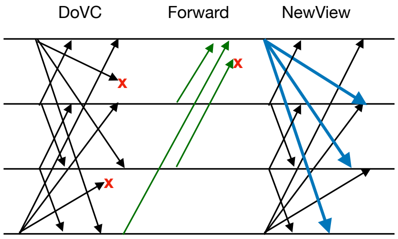


Fig. 1. Lock-step executions for $N = 4$ and $F = 1$.

The protocol is structured in three rounds, which form a phase. Fig. 1 shows an execution of ViewChange. In the first round, called DoVC, all replicas broadcast their current history (all-to-all communication). We assume authenticated communication channels between any two processes [??], i.e., the receiver of a message can be sure which process sent the message. All processes, including the leader, wait for $2F + 1$ messages. However, only $F + 1$ of these messages may come from correct replicas, which does not provide sufficient information to compute the most recent history of the system. The leader's ID is a function of the phase number (rotating leader), so the protocol uses the same leader in the next two rounds.

In byzantine systems, receiving one message is not sufficient to trust the received value, because byzantine processes might send different (possibly faulty) values to different processes. Therefore, to help the leader trust the received histories, in the second round, called *Forward*, the replicas that receive at least $2F + 1$ messages in the first round, forward these messages to the leader. The leader trusts an history $h_i$ sent by process $p_i$, if $F + 1$ processes acknowledge that they received $h_i$ from $p_i$. If the leader receives $2F + 1$ trusted values, based on them it computes, in the second round, the most recent trusted history in the system.

In the third round, called *NewView*, the leader broadcasts the previously computed history and the set of trusted pairs that the computation was based on. Also in the third round, because the leader might be byzantine, the other processes echo to the entire network the acknowledgments sent to the leader (in the second round). A process $p$ that receives the leader's message, checks that the latest history was correctly computed, using the messages from the other replicas to trust the values communicated by the leader.

```
1 val DoVC_mbox:List[(ProcessID, Hist)
2 val skip_next_round:Boolean
3 val history: Hist
4
5 new EventRound[Hist]{//Round: DoVC
6   def send(): Map[ProcessID,Hist] = {
7       broadcast(history)
8   }
9   def receiveInit(){
10      Progress.timeout(to)
11      Progress.catchUp(true)
12      Progress.sync(F+1)
13  }
14  def receive(sender: ProcessID) = {
15      if (mbox.size >= 2F+1)
            Progress.goAhead
16  }
17  def finishRound(mbox: List[(ProcessID,
        Hist)]) = {
18      if(mbox.size >= 2F+1)
19          DoVC_mbox = mbox
20      else // timeout
21          skip_next_round=true
22   }}
```

```
def receiveInit(){
    Progress.waitMessages
    Progress.catchUp(false)
    Progress.sync(0)
}

def receive(sender: ProcessID) = {
    if (mbox.size >= 2*F+1)
        Progress.goAhead
}
```

Fig. 4. Another message accumulator for the round in Fig. 3

Fig. 3. The 1st round of ViewChange (PBFT) in ReSync.

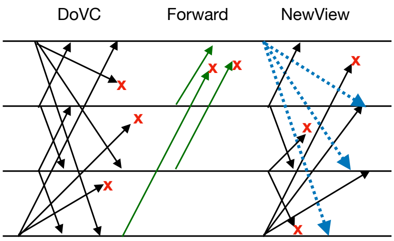


Fig. 2. Lock-step executions for $N = 4$ and $F = 1$.

The protocol executes these three rounds in a loop, because the execution of one phase does not guarantee that a majority of correct replicas are up to date, e.g., if the leader is byzantine. Fig. 2 shows an execution of a phase where although the leader is not byzantine, there are too many faults in the system to update sufficiently many replicas.

The algorithm is safe if less than a third of the processes are faulty and it is live, if eventually there is a good period where a quorum can communicate within itself and with the leader. This is an instance of *partial synchrony*, which states that eventually all correct processes can talk to each other.

*Encoding in ReSync.* The code of the first round, i.e., DoVC, is given in Fig. 3. The round implements four methods: send, defining the messages to be sent, `receiveInit` and `receive` computes the set of messages received in the round, i.e., the message accumulator, and `finishRound` that defines the round computation based on the received messages. In this round, all processes broadcast their current histories. Therefore, send uses the primitive `broadcast` and returns, at line 7, a (*key*, *value*) map, where the keys are all the process identities (the recipients) and the value is the `history` of the sender, i.e., a sequence of requests (type `Hist`). The method `send` is executed synchronously by all processes.

*Message accumulator.* In classic round-based models [?], the set of received messages is a subset of the sent messages, non-deterministically chosen by an adversarial environment. We enhance the round-based model with a message accumulator (specific to each round) that receives messages one by one. This gives the programmer the capabilities to react to a non-deterministic execution environment. The message accumulator is defined by a `Progress` object and two methods that use it, `receiveInit` (line 9) and `receive` (line 14). The main bottleneck of round-based models is an inefficient round-switch policy that requires synchronizing correct process. Using the methods of `Progress`, called *progress conditions*, the programmer guides the execution platform of ReSync, speeding up or slowing down the default round-switch policy. They are high-level constructs implemented by the execution platform. For example, using `Progress.timeout(to)` the programmer fixes in line 10 the time interval the execution engine waits for the messages of the current round, i.e., it waits for `to` milliseconds. Two other progress conditions are enabled.

Table 1. Network assumptions.

| | Lossy links | Reliable link |
|---|---|---|
| benign process crash | $F$=0 and UDP | $F$=0 and TCP |
| authenticated byzantine processes | $F$>0 and UDP | $F$>0 and TCP |

A message accumulator is designed taking into account the underling network. In this paper we consider different fault models, given in Table. ??. Regarding process failure, processes can crash (and not recover) or they can be byzantine. A byzantine process does not follow the protocol, it may send any messages up to forging the identity of other processes. We assume authenticated byzantine processes which cannot impersonate other processes, and this is done using digital signatures. The number of byzantine processes, denoted $F$, is a parameter of the program. If $F$ equals zero then the program tolerates only benign faults. The size of the network is fixed by the parameter $N$. For the link failures, we consider either lossy links, where messages can be lost, reordered, duplicated, or reliable links where every message sent it is eventually received. In practice lossy links are ensured by UDP and reliable links by TCP. The design of a message accumulator can take advantage of any other network properties the programmer is aware of, it is not restricted to the network assumptions discussed in this paper.

*Synchronization primitives.* ReSync has two synchronization primitives, `Progress.catchUp` and `Progress.sync`. They are implemented by the execution platform, for the types of network given in Table. ??. Note that a network with lossy links is a weaker network assumption than reliable links and it is one of the weakest network assumptions that one can consider. These primitives do not have a round-based definition for lossy links (`Progress.catchUp` does not have a round-based implementation even over networks that ensure reliable links).

The primitive `Progress.catchUp(true)`, Fig. 3 line 11, enables a process to switch rounds faster at runtime, as soon as it has received $F + 1$ messages with a higher round from different processes. It is used to resynchronize processes by advancing a slow process to the round of a faster correct one.

The primitive `Progress.sync(F+1)`, Fig. 3 line 12 requires that $F + 1$ *correct* processes switch rounds together, at runtime. Its implementation forces processes to receive at least $2F + 1$ messages from different processes for the current round $cr$, or a higher one, before moving to the next one, i.e., $cr + 1$. Note that the network might drop more than $F$ messages in a round (we recall $N = 3F + 1$). If the timeout expired and less than $2F + 1$ messages are received, the execution platform first increases the timeout but if this is not sufficient, it resends the messages of the current round (if the round does not use an all-to-all communication, the execution platform insert dummy messages to implement the synchronization requirement) [1]. `Progress.sync(F+1)` overrules the timeout constraints set by `Progress.timeout(to)`. Byzantine system cannot resynchronize (after an asynchronous period under partial synchrony) unless at least $F + 1$ processes are within the same round [?]. Therefore, even if `Progress.sync(F+1)` potentially slows down the round switch, it is necessary for byzantine consensus protocols.

For each round, the `Progress` object is configured in the method `receiveInit`. Messages are received one by one: the method `receive` takes as input a received message and implements the conditions under which this should be the last message received in the current round. Each process, in each round, has a fresh mailbox, that contains the queue of messages received (inputs to `receive`) by the process in that round. This mailbox is populated by the execution platform. The execution of the message accumulator starts with `receiveInit`, continues with multiple calls to *receive*, and stops when the progress conditions are met. The accumulator in Fig. 3 stops either due to `Progress.catchUp` or because `Process.sync(F+1)` holds and either the timeout expired or the control reached `Progress.goAhead` in Fig. 3 line15. Since `Process.sync(F+1)` is enabled, the message accumulator continues to collect messages until the progress condition is met even if the timeout expired. The messages accumulators of the same round execute asynchronously across processes.

*Round computation.* The method `finishRound` implements the computation done by a process in a round, based on the set of received messages. All (non-byzantine) processes execute the same code, however due to different mailboxes and branching, processes have different behaviors. `finishRound` is executed when a process stops receiving messages. The only computation done in the first round in ViewChange is to store the mailbox in the variable `DoVC_mbox`, in line 19. Across rounds, ReSync respects the classic round-based semantics: at the end of a round the mailbox of each process is purged and the computation moves in lockstep to the next round.

*Different message accumulators.* To solve byzantine consensus, it is sufficient that only some rounds force $F + 1$ correct processes to be synchronized. Therefore, in ViewChange at least one of the three rounds should enable `Process.sync`, but not necessarily the first one. If we consider the message accumulator from Fig. 3 with `Process.sync(0)`, then the message accumulator might stop due to an expired timeout, before receiving $2F + 1$ messages. Therefore when `finishRound` is executed, line 21 is reachable and `skip_next_round` is true. In ViewChange, processes that do not receive $2F + 1$ messages in the first round move on to the next phase. In the round-based structure processes cannot skip rounds. However, they are not forced to send messages or do any computation in a round. The flag `skip_next_round` is used to encode the behavior of these processes, which go through the second and third round of the current phase without sending any messages or doing any computation.

Let us consider the message accumulator in Fig. 4 which uses `Progress.waitMessages` instead of `Progress.timeout`. The semantics of `Progress.waitMessages` states that calls to *receive* are made until `Progress.goAhead` is reached. Since `Progress.catchUp` is disabled, the only way for a process to switch round is if $2F + 1$ messages are received. This message accumulator can get stuck, if the program is executed over a network that provides unreliable communication. However, if

[1] The implementation of `Progress.sync(F+1)` is possible only for partially synchronous networks.

```
 1 // Commit request
 2 new EventRound[Int]{
 3   def send(): Map[ProcessID,Int] = {
 4     if (id == leader) {
 5       broadcast(transactionID)
 6     } else { Map.empty }
 7   }
 8   def receiveInit = {
 9     Progress.waitMessages
10     Progress.catchUp(false)
11   }
12   def receive(sender: ProcessID,
          payload: Int) = {
13     Progress.goAhead
14   }
15   def finishRound(mbox:
          List[(ProcessID, Int)]) = {
16     commit = check(localState, mbox)
17 } }
```

```
 1 // Vote (Yes/No)
 2 new EventRound[Boolean]{
 3   def send(): Map[ProcessID,Boolean] = {
 4     Map( leader -> commit )
 5   }
 6   def receiveInit = {
 7     if (id != leader) Progress.goAhead
 8     else  Progress.waitMessage
 9   }
10   def receive(sender: ProcessID, payload:
         Boolean) = {
11     if(!payload || mbox.size == n)
          Progress.goAhead
12   }
13   def finishRound(mbox: List[(ProcessID,
         Boolean)]) = {
14     if (id == leader) {
15       decision = head(mbox)._2
16 } } }
```

Fig. 5. Voting phase of the *Two phase commit* protocol in ReSync.

executed with reliable communication between correct processes, e.g., TCP, it is an implementation (in the new round model) of the synchronization primitive `Process.sync(F+1)`.

The message accumulator can also depend on the value in the payload of messages. Let us look at the two-phase commit protocol. Fig. 5 presents the first two rounds — i.e., the first phase — of the protocol. It is an atomic commitment protocol, executed by $n$ benign processes (that is all processes follow the protocol). This protocol assumes crash recovery, i.e., the nodes have stable storage and they can resume operation after a crash. Therefore, the processes block until they receive the messages they expect.

In the example, the transaction identifier is an integer that the leader proposes (at line 5) to all processes. Processes blocks until they receive the message from the leader (lines 9 and 13). Each process checks locally whether it can locally commit the transaction (without violating the consistency of the database) at line 16. In the second round, on the right, processes send (at line 4) their vote to the leader. The two-phase commit protocols says that a transaction can commit only if all the participants agree. Therefore, the leader wait for $N$ "yes" votes or a single "no" vote (line 11). 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.

In our example, we do not give concrete values for timeouts. The actual timeout value might vary across rounds as they are influenced by the amount of computation required by handling messages. For example, if we consider networks with non-authenticated communication channels, messages are signed using public encryption policies, and the signatures are checked by the recipients. The timeout depends on the size of the information to be signed and on the signature algorithm. By choosing a timeout that takes only the network delay into account, the round switch may happen too fast, and processes could miss messages that are delivered by the network.

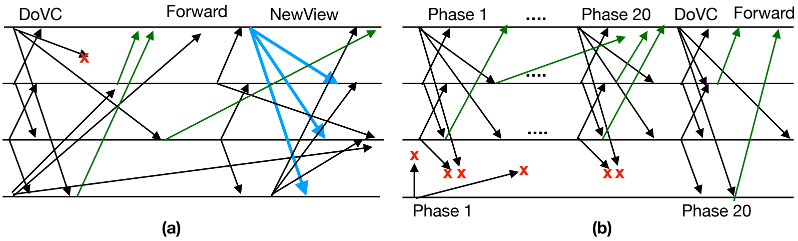


Fig. 6. Runtime executions for $N = 4$ and $F = 1$.

space between the two runs

*Runtime executions.* Fig. 6(a) and Fig. 6(b) show two runtime executions of ViewChange in ReSync. At runtime, processes are not all in the same round (due to the asynchronous nature of the network). Therefore, processes receive messages from different rounds out of order. The runtime discards the late messages and stores the messages from the future rounds. It delivers only the messages for the current round. Since `Progress.sync(F+1)`, at runtime, a process will not finish DoVc when the timeout expired, unless it has received at least $2F + 1$ messages for the current round (or a higher one). However, a process finishes the other two rounds independently of the number of received messages.

Not all processes are forced to wait for $2F + 1$ messages. The progress condition `catchUp` makes processes jump to a future round, if they received more than $F + 1$ messages for that round (or a higher one). For example, in Fig. 6(b), the bottom process is slowed down in the round DoVc of the first phase and jumps to round DoVC of the 20th phase, when it receives $F + 1$ messages for it. This process is not forced to wait for $2F + 1$ messages as the reception of $F + 1$ messages for a higher round guarantees that there were (other) $F + 1$ correct processes synchronized. We show that the runtime executions are refinements of the lockstep executions: processes at runtime go through the same sequence of local states as in the round-based semantics. For example, the execution in Fig 6(a) is a refinement of the execution in Fig 1.

## 3 SYNTAX OF RESYNC

The syntax of an ReSync program is given in Fig. 7. A program defines the code executed by one process. In ReSync all processes execute the same code. However, within a round the programmer can 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 that are used to communicate with other distributed programs. The interface has inbound operations, for example to get 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* has variables declaration, an initialization function, the main distributed computation part, and sequential pure function. All variables are local to a process. The communication between processes is done exclusively by message passing.

*Variables.* All ReSync programs have several predefined variables: $N$ the total number of processes, $F$ the number of byzantine processes (both parameters of the program), and $r$ the round number. These variables are read-only.

| | | | |
|---|---|---|---|
| *program* | ::= | *interface* | EventRound$_{\mathsf{M}}$ |
| | | *body* | `mbox`: $List[P \times \mathsf{M}]$ |
| *body* | ::= | $\mathit{var\_decl}^*$ | `Progress`: `ProgressC` |
| | | $\mathit{pure\_fun}^*$ | `send`: $() \to [P \mapsto \mathsf{M}]$ |
| | | *init* | `receiveInit`: $() \to ()$ |
| | | *phase* | `receive`: $(s : P,\ m : \mathsf{M}) \to ())$ |
| *phase* | ::= | $\mathit{round}^+$ | `finishRound`: $(\mathit{mbox}) \to ()$ |

Fig. 7. ReSync abstract syntax.

*Initialization.* Each program implements an `init` method, which configures the initial state of a process: the initial values are obtained using the inbound interface operations. The method `init` takes as input interface objects, it does not return any values. It modifies the process's variables without sending or receiving messages. Implicitly, `init` initializes the round number $r$ to zero.

*Rounds.* The distributed computation part is structured into a fixed sequence of rounds, called *phase*. Each round is an instance of an abstract class `EventRound`, whose structure is described in Fig. 7. Rounds are parameterized by a payload type $\mathsf{M}$ and have two attributes: 1) `mbox`, of type $List[(P, \mathsf{M})]$, is a queue of messages, where $P$ is the process type, and a progress object, `Progress`, where `ProgressC` is the class of progress conditions. In the code snippets, we use process identifier (`ProcessID`) for $P$. For the semantics, these two types are the same. 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 `receiveInit` and `receive` is optional. If `receiveInit` and `receive` are implemented, the round is called an *open round*, otherwise it is called a *closed round*. In closed round `Progress.catchUp` and `Progress.sync(F+1)` are enabled by default.

The type of messages exchanged in different rounds might differ, however within a round, all messages have the same payload type. The control flow of a round is a sequence of method calls, starting with `send`, followed by `receiveInit` and a loop of calls to `receive`, if the latter two are present, and lastly `finishRound`. The round variables `mbox` and `Progress` 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., $[P \mapsto \mathsf{M}]$, 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). The message accumulator stops, when a certain condition holds. Next, the method `finishRound` is executed and the control switches to the next round.

The algorithm designer implements the message accumulator by defining `receiveInit` and `receive` and controls when the accumulator terminates using different *progress conditions*. A progress condition is a method or an attribute of a `ProgressC` object. They impose different conditions on when the accumulator may stop receiving messages:

- `Progress.timeout(to: Milliseconds)`, the accumulator is enabled to end receiving messages after a timeout expired, that is `to` time units after the round started;
- `Progress.goAhead`, the accumulator is enabled to end as soon as the method `goAhead` is called;
- `Progress.waitMessages` is a flag; when set, it means that there is no timeout to stop the message reception.

- `Progress.catchUp(b: Boolean)` allows or disallows the execution platform to (re-)synchronize processes under a partial synchrony assumption;
- `Progress.sync(k: Int)` is a global synchronization primitive that constraints the message accumulators of $k$ correct processes to end synchronously.

The method `receiveInit` configures the progress conditions. The programmer may implement different progress conditions for different processes in the same round by branching. The method `receive` defines the `mbox` and checks if the message accumulator should terminate. It takes as input an incoming message, that is, a pair $(sender, value)$, where *sender* is the process which sent the message , 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. In `receive` the programmer uses `Progress.goAhead` to terminate the message accumulator when a condition on the mailbox computed so far holds.

When the methods `receiveInit` and `receive` are not implemented, the mailbox is a non-deterministic subset of the sent messages.

*Round computation.* In the method `finishRound` the programmer defines how the process state changes at the end of a round. The method `finishRound` takes as input the queue of messages received, i.e., takes as input `mbox`. It may call outbound operations from the interface, to communicate the results of the computation to a client.

## 4 LOCK-STEP SEMANTICS OF RESYNC

Given a program $\mathcal{P}$, all (non-byzantine) process execute, in a loop, the sequence of rounds defined in the program. The semantics of a program is given by the synchronized parallel composition of the transition systems associated with each process. We denoted by $[\![\mathcal{P}]\!]$ the set of executions defined by the transition system in Fig. 8, Fig. 9, Fig. 10, and Fig. 11.

The execution progresses in lockstep, 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 `receiveInit`, `receive`, and the network transitions are interleaved across processes within the round boundaries. Locally, a process executes a sequence of calls to round methods corresponding to

$$init;\ \big(send;\ \big((receiveInit;\ \ receive^*) \mid NondetNet\big)\ ; finishRound\big)^*$$

where *NondetNet* represents a network transition. All processes execute the same sequences of rounds.

The state $S$ of a ReSync program over the set of processes $P$ 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;
- $s \in [P \longrightarrow Vars \cup \{lstep\} \longrightarrow \mathcal{D}]$ stores the local states of the processes, $Vars$ is the union of all variables, $\mathcal{D}$ is the domain of values, and *lstep* is a special variable taking a value in $\{Send, Recv, Net, Fin\}$ and ;
- $r \in \mathbb{N}$ 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 ordered set of delivered messages, all the messages in *sms* and *dms* have a different pair of sender/receiver, i.e., a process can only send one message to another process in each round.

To model correct, crashed, and byzantine processes we use the sets *CoP*, *Cr*, and *ByzP*. *CoP* is the set of correct processes, *Cr* is the set of crashed processes, and *ByzP* contains the byzantine processes. These sets form a partition of the processes, i.e., $P = CoP \uplus Cr \uplus ByzP$. $N$ and $F$ takes

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

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

$$\textsc{Send-Closed}\quad \frac{\forall p \in Pr.\ s(p) \xrightarrow{m_p = \mathsf{phase}[r].\mathsf{send}(), \emptyset} s(p) \qquad sms = \bigcup_{p \in P} \{(p, m, q) \mid q \in P \land (q, m) \in m_p\}}{\langle Send, s, r, \emptyset, \emptyset\rangle \overset{\emptyset, \{\mathsf{send}_p(m_p) \mid p \in P\}, \emptyset}{\Longrightarrow} \langle Net, s, r, sms, \emptyset\rangle}$$

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

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

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

Fig. 8. The global semantics of ReSync. A transition $s(p) \overset{\mathsf{phase}[r].m, o}{\rightarrow} s'(p)$ denotes that $p$ executes, in local state $s(p)$, the method $m$ of the EventRound phase$[r]$. The execution produces observable events $o$, corresponding to calls to methods from the interface.

$$\textsc{Drop}\quad \frac{sms = sms' \uplus \{m\}}{\langle s, sms, dms\rangle \overset{Dr}{\longmapsto} \langle s, sms', dms\rangle} \qquad \textsc{Crash}\quad \frac{s'(p) = \bot \qquad \forall q.\ q \neq p \Rightarrow s(q) = s'(q)}{\langle s, sms, dms\rangle \overset{Cr}{\longmapsto} \langle s', sms, dms\rangle}$$

$$\textsc{Deliver}\quad \frac{sms = sms' \uplus \{(p, v, q)\} \qquad (p \in ByzP \lor v = v') \qquad dms' = dms \cup \{(p, v', q)\}}{\langle s, sms, dms\rangle \overset{Dv}{\longmapsto} \langle s, sms', dms'\rangle}$$

Fig. 9. Network transition on the message pools.

the values $N = |P|$ and $F = |ByzP|$. For crashed processes, we set their local state to the special $\bot$ value, i.e., for a store $s$ we have $Cr = \{p \in P \mid s(p) = \bot\}$. Crashed processes never recover. $Pr$ is a shorthand the set of running processes, i.e., $Pr = P \setminus Cr$.

The rules for the round structure where the processes work in lockstep are shown in Fig. 8. Later, we will see additional rules for the network (Fig. 9), closed rounds (Fig. 10), and open rounds (Fig. 11). Global lockstep transitions are written as $S \overset{E,I,O}{\Longrightarrow} S'$ where $S$, $S'$ are states, $E$ 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 ReSync program can only observe the transitions with labels in $O$. Local transitions of a process are written with $\longrightarrow$ and transition of the network are denoed with $\longmapsto$.

*Init.* Initially the state of the system is undefined, denoted by $*$. The first transition of the system, captured by the rule Init in Fig. 8, defines the initial state of the protocol by calling `init` on all processes. All processes execute in lockstep (synchronously) `init`. The method `init` does not

$$\textsc{Closed-NetworkSteps}\quad \frac{\langle s,\, sms,\, dms\rangle \overset{N}{\longmapsto} \langle s',\, sms',\, dms'\rangle}{\langle Net,\, s,\, r,\, sms,\, dms\rangle \overset{N,\emptyset,\emptyset}{\Longrightarrow} \langle Net,\, s',\, r,\, sms',\, dms'\rangle}$$

$$\textsc{Closed-Mailbox}\quad \frac{\begin{array}{c}\forall p \in Pr.\ s(p)(lstep) = Net \wedge s'(p)(lstep) = Fin \\ \wedge s(p)|_{\mathsf{Vars}} = s'(p)|_{\mathsf{Vars}} \wedge s'(p)(\mathsf{mbox}) = [(q,m) | (q,m,p) \in dms]\end{array}}{\langle Net,\, s,\, r,\, sms,\, dms\rangle \overset{\emptyset,\emptyset,\emptyset}{\Longrightarrow} \langle Fin,\, s',\, r,\, sms,\, dms\rangle}$$

Fig. 10. Network transitions for closed rounds.

return a value and defines the local state of a process using inbound operations from the interface. The global transition Init emphasizes the inbound operations called by `init`, denoted $o_p$, where $o$ is the operation's name and $p$ is the process executing it. Init sets the round counter to 0, the message pools are empty (there are no messages in the system), and globally the system transitions to a *Send* state, i.e., $gstep = Send$. Rounds are executed in a loop. Next, we define the semantics of a round.

*Send.* In every round, the first transition is a global one, which executes locally in lockstep the method send on all processes. Its semantics is formalized by the rules Send-Open and Send-Closed in Fig. 8. Only one of the two rules applies in a round. Send-Open is applied when `init` and `receive` are defined. The control label *gstep* becomes *Recv*, i.e., the control goes to the implementation of a message accumulator. Send-Closed 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*, otherwise *gstep* equals *Recv*. The method send has no process local side effects and returns a map from receivers to payload values, i.e., the messages sent by the process that executes send. The rules Send-Closed, resp. Send-Open add the messages sent by processes to the pool of sent messages, *sms*. The messages in *sms* are triples of the form (sender, payload, recipient), 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.

After sending messages the control passes to the network that non-deterministically decides which messages to deliver and which to alter.

add duplication rule.

dz: duplication is never visible to the programmer,

*Network transitions.* The pool of messages *sms* and the lists of delivered messages *dms* are managed by a special *network* process that modifies the pool of messages *sms* and *dms* (for simplicity the rules omit its identity), and in the case of open rounds controls also the progress conditions. The rule Deliver, in Fig. 9, captures how the network takes a message $m$ from the message pool *sms* and puts it into the delivery set *dms*. For the byzantine case, we model the malicious behaviours by changing the delivered messages to arbitrary values. This way of modeling means we can keep uniform rules for sending and processing messages. 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.

*Message reception in closed rounds.* When `receiveInit` and `receive` are missing (they are not redefined), after Send-Closed, the rule NoRecv-NetworkSteps is applied, a nondeterministic number of times, followed by the rule NoRecv-Mailbox, from Fig. 10. 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.

**Progress transitions**

$$\textsc{Progress.goAhead}\quad \frac{s'(p)(goAhead)}{s(p) \xrightarrow{\substack{\texttt{receiveInit}_p() \\ \texttt{receive}_p(q,v)}} s'(p),} \qquad \textsc{Progress.waitMessages}\quad \frac{\neg s'(p)(to)}{s(p) \xrightarrow{\substack{\texttt{receiveInit}_p \\ \texttt{receive}_p(q,v)}} s'(p)} \qquad \textsc{Progress.sync}\quad \frac{s'(p)(sync)}{s(p) \xrightarrow{\substack{\texttt{receiveInit}_p \\ \texttt{receive}_p(q,v)}} s'(p)}$$

$$\textsc{Progress.catchUp(true)}\quad \frac{s'(p)(catchUp)}{s(p) \xrightarrow{\substack{\texttt{receiveInit}_p \\ \texttt{receive}_p(q,v)}} s'(p)} \qquad \textsc{Progress.timeout}\quad \frac{s'(p)(to)}{s(p) \xrightarrow{\substack{\texttt{receiveInit}_p \\ \texttt{receive}_p(q,v)}} s'(p)}$$

**Accumulator transitions**

$$\textsc{Acc-Init}\quad \frac{s(p)(lstep) = Recv \qquad s(p) \xrightarrow{\texttt{receiveInit}_p} s'(p) \qquad s(p)|_{\texttt{Vars}} = s'(p)|_{\texttt{Vars}}}{\langle s(p), sms, dms\rangle \xrightarrow{\{\texttt{receiveInit}_p\},\emptyset} \langle s'(p), sms, dms\rangle}$$

$$\textsc{Acc-Recv}\quad \frac{\begin{array}{ccc} \neg s(p)(goAhead) & dms = dms' \uplus \{(q,v,p)\} & s(p) \xrightarrow{\texttt{receive}_q(q,v)} s'(p) \\ s(p)(lstep) = s'(p)(lstep) = Recv & s(p)|_{\texttt{Vars}} = s'(p)|_{\texttt{Vars}} & s'(p)(\texttt{mbox}) = (q,v) :: s(p)(\texttt{mbox}) \end{array}}{\langle s(p), sms, dms\rangle \xrightarrow{\{\texttt{receive}_p(q,v)\},\emptyset} \langle s'(p), sms, dms'\rangle}$$

$$\textsc{Acc-End}\quad \frac{\begin{array}{c} s(p)(lstep) = Recv \qquad (s(p)(GoAhead) \lor s(p)(to) \lor s(p)(catchUp)) \\ s(p)|_{\texttt{Vars}} = s'(p)|_{\texttt{Vars}} \qquad s'(p) = Fin \qquad \neg s'(p)(GoAhead) \qquad \neg s'(p)(to) \end{array}}{\langle s(p), sms, dms\rangle \xrightarrow{\emptyset,\emptyset} \langle s'(p), sms, dms\rangle}$$

**Global transitions**

$$\textsc{Acc-NetworkStep}\quad \frac{\langle s, sms, dms\rangle \overset{N}{\longmapsto} \langle s', sms', dms'\rangle}{\langle Recv, s, r, sms, dms\rangle \xRightarrow{N,\emptyset,\emptyset} \langle Recv, s', r, sms', dms'\rangle} \qquad \textsc{Acc-ProcessStep}\quad \frac{\langle s, sms, dms\rangle \xrightarrow{I,\emptyset} \langle s', sms', dms'\rangle}{\langle Recv, s, r, sms, dms\rangle \xRightarrow{\emptyset,I,\emptyset} \langle Recv, s', r, sms', dms'\rangle}$$

Fig. 11. Message accumulator transitions.

*Message accumulator (message reception in open rounds).* The implementation of the message accumulator starts after sending messages (after apllying Send-Open), by updating the local variable *lstep* to *Recv* (see rule RecvStart in Fig. 8). To compute the mailbox of an open round, process executes first `receiveInit` (see rule Acc-Init in Fig. 11) and afterwards a loop of calls to `receive` (see rule Acc-Recv in Fig. 11). Calls to `receive` stop when the progress conditions configured in `receiveInit` are met (see rule Acc-End).

*ReceiveInit.* The progress conditions of a round are set in `receiveInit`. To formally define them we use auxiliary boolean local variables: *goAhead*, *catchUp*, *to*, and *sync*. Because the semantics is given at the lockstep level and the progress conditions are hints to control the runtime algorithms, the progress condition does not have much impact on the semantics. For instance, the timeout is abstracted as a boolean value and the accumulator can non-deterministically continue receiving messages or end.

*Receive.* The method `receive` when executed by process $p$, takes as input a delivered message in the set $dms$ (see rule Acc-Recv) and adds it to the mailbox of the current round. In each round, at most one message is received from any process, i.e., duplicates if present are eliminated. Another call to *receive* follows only if the progress conditions are not met, see rule Acc-End in Fig. 11. The message accumulator ends because either the catchUp condition holds, or the synchrony condition holds together with either the goAhead or the timeout condition (see rule Acc-End). Note that the catchUp, synch, and timeout progress conditions are in this lockstep semantics under the control of the network, and the protocol assumes the network respects their semantics. The execution platform provides an implementation of these primitives.

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. 8.

*Round-based Algorithm transition.* In a FinishRound transition (Fig. 8), all processes execute synchronously (without any interference) the method `finishRound` of the current round. Locally, on each process $p$, the set of received messages $\mathtt{mbox}_p$ (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.

## 5 RUNTIME OF RESYNC

We introduce an execution platform for ReSync over asynchronous networks. The runtime executions are refinements of the lockstep executions from Section ??. For closed rounds, if the underlying network is partially synchronous, the runtime eventually synchronizes correct processes, ensuring that they can communicate reliably with each other.

### 5.1 Runtime semantics

The *runtime algorithm* in Listing 1 defines the code executed at runtime by one process. Listing 1 shows an extract from the actual implementation (which has lines of code) that highlights the control structure of the runtime algorithm. Given a ReSync program $\mathcal{P}$, the set of runtime executions $[\![P]\!]_{Run}$ is defined by the asynchronous parallel composition of $N$ transitions systems corresponding to Listing 1.

The runtime algorithm defines a wrapper for the ReSync code. It has the following structure: declaration of the variables that keep track of the timeout, the round number, and the buffer of delivered messages (line 6), a few auxiliary methods (lines 15-29), and a big while loop where every iteration corresponds to the execution of one round (lines 32-59). All processes go through all rounds in order. Each loop iteration is further split into initialization (lines 37-40), send (line 34), message accumulator (lines 42-54), and finishing the round (lines 56-58). The message accumulator is a loop where at each iteration one message is received and processed. The exit of this loop triggers a round switch.

The runtime uses `phase`, the sequence of *len* rounds defined in ReSync, and the variable `currentRound` holding the current round number. In Fig. 1, `phase[currentRound]` is the $\mathtt{k}^{th}$ EventRound defined in the ReSync program, where $k$ equals `currentRound%len`. The runtime uses the implementations of `send`, `receiveInit`, `receive`, `finishRound` given in ReSync by calling `phase[currentRound].send`, etc. Each call is wrapped by `checkProgress` that implements the semantics of the progress conditions.

Listing 1. ReSync Runtime Algorithm

```
// Global constants
id: Pid; N, F: ℕ //own process id, number of processes, malicious processes
round: Array[EventRound_M] //Program
// State shared with the program
Progress: ProgressC
mbox: List[Msg] = [] // mailbox of the current round
// Internal state
currentRound, nextRound: ℕ = 0
roundStart, timeout: Time
strict, inSync: 𝔹 = false // flags to control the progress
maxRound: Map[Pid, ℕ] = Map[for (p <- Pid) yield (p, 0)] // keep the max round seen for each process
pendingMessages: Map[ℕ, Msg] = Map[for (n <- ℕ) yield (n,∅)] // buffered messages (asynchrony)

// Auxiliary Methods
def processReceive(message: Msg ) { // Receiving a message
    if (∀ m ∈ mbox. m.sender ≠ message.sender) { // check duplication (UDP or byzantine)
        mbox = message :: mbox
        round[currentRound].receive(message.sender, message.payload)
        checkProgress()
}   }
def checkProgress() {  // Update timeout and strict according to Progress
    if (Progress.isTimeout) { timeout = Progress.timeout; strict = !Progress.catchUp }
    else if (Progress.isWaitMessage) { timeout = ∞; strict = true }
    else if (Progress.isGoAhead) { strict = false; nextRound = max(nextRound, currentRound + 1) }
    if (Progress.syncNb > 0) {
        if (maxRound.values.filter( _ >= currentRound).size >= Progress.syncNb + F) { strict = false }
        else { timeout = ∞; strict = true }
}   }
def catchUpTo() = maxRound.values.sorted[N - F - 1] // highest round excluding byzantine processes

//Execute the program, each iteration of the loop corresponds to one round
while(true) {
    // SEND
    for( (dest, payload) <- round[currentRound].send() )
        Network.sendTo(dest, Msg(id, currentRound, payload))
    // RECEIVE INIT
    strict = false; mbox = [] // clean
    roundStart = currentTime() // set time
    round[currentRound].receiveInit()
    checkProgress()
    // RECEIVE
    for ( message <- pendingMessages[currentRound] ) processReceive(message) // deliver buffered msg
    while ((nextRound == currentRound ∨ strict)) {  // not enough messages received
        message = Network.receiveWithTimeout(roundStart + timeout - currentTime())
        if (message == null) { // timeout
            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) { processReceive(message) }
            else { pendingMessages[message.round].add(message) // buffer and
                   nextRound = max(nextRound, catchUpTo()) } // check if catching-up is needed
    } }
    // FINISHROUND
    round[currentRound].finishRound(mbox)
    currentRound += 1; maxRound[id] = currentRound
    nextRound = max(nextRound, currentRound)
}
```

Message reception is implemented using the method `receiveWithTimeout` in line 44 that either returns a message or `null` if the timeout (given by the ReSync program) expired. If the progress condition `waitMessages` is active, `receiveWithTimeout` is called with an infinite timeout. The messages delivered to a process are 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 $r$, $r$'s message buffer will contain messages from different rounds. The 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 buffer `pendingMessages` 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 `receiveInit` is to set a timeout, enable `catchUp` and enable the synchronization of $F + 1$ processes. The default behavior of `receive` is to add a 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. The function `checkProgress` updates the local variables that gate a round switch at line 43. This function is called after the reception of a message (line 51) and in the beginning of every round with the call of `receiveInit` (line 40) but also on the buffered set of messages for the current round (line 42) received while the system was operating in a lower round.

*CatchUp algorithm.* This primitive enables a process to *jump over* over some rounds at runtime, so that it can *catch-up* with processes that made quicker progress. Catch-up is a mechanism used by many asynchronous systems, e.g, Paxos, ViewStamped, PBFT. Implementing this primitive requires breaking communication-closure which is possible only at execution time.

In the benign case, i.e., $F = 0$, 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. Note that catch-up is useful to synchronize processes (with the faster ones) the under lossy links, because messages of the current round might be lost, but also under reliable links due to message reorder (see Table **??**). The implementation of catch-up uses two variables `nextRound`, ranging over round numbers, and `strict` which is a boolean mirroring `Progress.catchUp`.

In the byzantine case, i.e., $F > 0$, it is not sound for a process to catch-up upon a single message reception, because a Byzantine process may send arbitrary round numbers. Therefore, 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 29) in the array `maxRound`, and jumps to the $F$ + 1st highest one (at least one correct process made progress until that round).

In case of a jump, the message accumulator of the current round stops, and `finishRound` is executed for the current and all the other rounds, up to `nextRound`.

*Synchronization algorithm.* Although all protocols may use `Progress.sync(k)`, except for two phase commit (where $k = N$), this primitive is useful mainly in byzantine protocols, where $k$ equals $F + 1$. It ensures that $k$ correct processes, i.e., non-byzantine and not-crashed, are synchronously switching at runtime the respective round.

dz: i don't understand the 2PC remark

The implementation of `Progress.sync(k)` forces the runtime to wait for $F + k$ messages sent by different processes for the current round or any other higher round before terminating the current message accumulator, and do a round switch. This is done using the `maxRound` map which stores the highest round number received from each process. Over networks with reliable links, two timeouts ensure that between round switches the waiting time is long enough to let correct process exchange messages. However, over networks with lossy links timeouts are an underestimation. The transport layer is responsible for resending the messages of the round, and therefore, the synchronization algorithm an infinite timeout assuming the transport layer will eventually deliver enough messages. The implementation of this synchronization primitive is possible in the round-based model only under reliable networks, i.e., TCP. Resending the messages of a round breaks the round structure, which requires the number of sent messages to be bounded by the state of the process and the number of processes in the network. Also, over networks with lossy links it is necessary to consider messages from higher rounds. A message that is from a round $fr$ greater than the current round $cr$, witnesses that the process which sent it was at some point in $cr$, therefore it will be counted among the $2F + 1$ messages (from different processes) required to validate the synchronization barrier. This behavior it is not possible implementable directly in the round model.

The runtime uses an auxiliary method `processReceive` to check that only one message per process per round is delivered. This deals with malicious processes which may send multiple messages and link faults, e.g., UDP may duplicate packets.

## 5.2 Runtime correctness

At runtime, the executions are asynchronous, i.e., processes may be in different rounds at the same time, and processes may receive messages coming from rounds that are different from their current one. We show that these runtime asynchronous executions are indistinguishable from the ReSync lockstep semantics.

*Definition 5.1 (Indistinguishability).* Two executions $\pi$ and $\pi'$ 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.

Next, by *send*, *receiveInit*, *receive*, and *finishRound*, we refer to the execution, at runtime, of the wrappers of those respective methods in ReSync program.

Theorem 5.2 (Correctness). *Given a ReSync program $\mathcal{P}$, for every execution $ae \in [\![(\mathcal{P})]\!]_{Run}$, there exists an indistinguishable execution $se \in [\![\mathcal{P}]\!]$ with respect to the actions $C = \{init, send,$ receiveInit, $receive, finishRound\}$, 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 ReSync program. Thus, reduction arguments [**???**] due to communication closure allow 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 [**?**], 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 lead to stuttering invisible events, which we may drop and maintain indistinguishability. □

Theorem 5.2 implies that the specifications closed under indistinguishability are preserved by the execution platform. These specifications (also called local properties in [**?**]) form an important class, which includes consensus, k-set agreement, and leader election.

For networks that are partially synchronous we prove a stronger result. Partially synchrony is defined over the physical model of the network, with immediate consequences over the runtime executions, but also over the high-level round-based executions. Let $T(m)$ be the time it takes between the moment a message $m$ is added to the send pool *sms* and the moment it either gets dropped by the network or it is delivered to its recipient. A (physical) network is *partially synchronous* if there exists a $\Delta$, and a global stabilization time $GST$, such that the transmission delays of all messages $m$ sent after $GST$ from a correct processes to a correct process satisfy $T(m) < \Delta$. The original definition in [?] assumes also that a bound on the relative speed of processes holds only after $GST$. However, on modern architectures this bound holds during the entire computation. Note that before $GST$ arbitrarily many messages might be dropped.

In the round-based model, the existence of GST means that from some round on, called $GSR$, all messages sent by correct processes to correct processes are received. Prior to $GSR$ arbitrarily many messages may be dropped. Given a program $\mathcal{P}$, the set of partially synchronous lockstep executions of $\mathcal{P}$ is a subset of $[\![\mathcal{P}]\!]$, consisting only in those executions where there exists global synchronization round $GSR$.

Theorem 5.3 (PS Correctness – Closed). *Given a ReSync program $\mathcal{P}$ with closed rounds, for every execution $ae \in [\![(\mathcal{P})]\!]_{Run}$ over a partially synchronous network, there exists an indistinguishable partially synchronous execution $pse \in [\![\mathcal{P}]\!]$ w.r.t. the common actions $C = \{send, receiveInit, receive, finishRound\}$, that is, $ae \sim_C pse$.*

The proof of Theorem 5.3 follows similar ideas as the proofs in [?]. In the following, we emphasize a few crucial differences that make the execution platform of ReSync practical. A distributed clock is constructed [?], which ensures that at least $F + 1$ correct processes have closely synchronized local estimates of the clock. The distributed clock has two purposes: (i) it ensures that after GST all correct processes will quickly resynchronize and (ii) a process uses the clock ticks to determine how long to wait for the messages of a round. In [?], at every process, the steps of the distributed clock algorithm are interleaved with the steps of the consensus protocol that is executed. During a round of the consensus algorithm, the clock ticks multiple times, as the clock is used to burn time and slow down fast processes (to wait sufficiently long for messages). Each tick is implemented by an instance of a distributed clock synchronization scheme, which involves all-to-all communication. In practice this would flood the network with many messages and slow down the consensus computation (due to interleavings of the clock with the algorithm).

Instead of using a distributed clock to timeout messages, the runtime of ReSync uses a hardware clock, and it implements a distributed clock only to synchronize the round start, that is, it uses only one clock value per round, contrary to many clock ticks in [?]. By replacing many distributed clock ticks with a hardware clock ReSync's runtime gives the programmer the possibility to use timeouts that 1) estimate the time it takes for the slowest correct process to resynchronize with the fastest correct process and 2) estimate the time to process and transmit a message.

*Definition 5.4 (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 observable actions of a ReSync program are calls to the interface methods. The runtime executions and the lockstep executions are w.r.t. the interface actions. In [?] sequentially consistency is proven equivalent with observational refinement for commutative clients, i.e., when the interface operations commute. This proof is adapted straightforward to indistinguishability.

Corollary 5.5. *Given a program $\mathcal{P}$ the runtime semantics of $\mathcal{P}$ observationally refines the ReSync 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 $[\![(\mathcal{P}[PS])]\!]_{Run} \triangleright_O [\![\mathcal{P}[PS]]\!]$.*

### 5.3 Runtime Implementation Details

The runtime implements the algorithm in Listing 1. We discuss in this section aspects related to the message structure and 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 needs to be provided by the user and properly hardened.

*Self messages bypass.* Sending a message to self would be wasteful. ReSync 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 aspects.* One important aspect when implementing the runtime is keeping the maximal amount of used memory bounded. Bounding the memory is essential when considering malicious attackers. Since we store messages that are supposed to be delivered later, in the buffer `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.

Reallocating all the data structures required by the runtime, e.g., `maxRound`, `pendingMessages`, with each new program execution, causes an important performance loss. Therefore, we pool and reuse the objects with a complex internal structure as well as the memory buffers for sending and receiving messages.

*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 EXPERIMENTAL EVALUATION

We have implemented ReSync in Scala on top of the PSync [?] codebase and evaluated it experimentally. The goal of the experiments is to 1) compare the performance ReSync with largely-used distributed systems 2) compare the expressiveness and performance of ReSync with other domain specific languages, especially with PSync, and 3) compare different ReSync implementations of the same theoretical algorithm.

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 the runtime of ReSync allocates most resources at the beginning, we report averages over runs of 5 minutes. The implementation is available at https://github.com/dzufferey/psync and a exlpanation on how we run the experiments is available at https://github.com/dzufferey/resync_oopsla20_artifact. The ReSync model is agnostic to the transport layer. We support TCP and UDP in the benign case. Authentication for Byzantine protocols is implemented using TLS on top of TCP[2].

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

This also protects from a large class of Sybil and replay attacks. Being able to precisely identify the sender and only receive from expected senders is necessary for the integrity of the model.

### 6.1 Throughput evaluation

We perform two experiments comparing the throughput of consensus algorithms in ReSync against widely used systems solving consensus. The first experiment looks at the case of benign faults while the second one compares Byzantine fault-tolerant implementations.

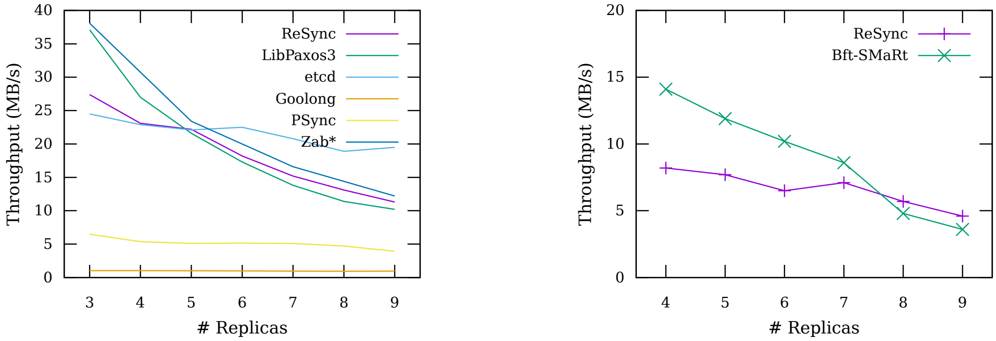
*Benign consensus.* ReSync is evaluated on an iterated version of Paxos (multi-Paxos), that uses an adaptation in ReSync of round-based version of Paxos given by [?], called LastVoting. To this, we implemented a message accumulator where the leader progresses as soon as it receives a majority of votes and the acceptors progress after receiving a message from the leader. The algorithm is used in a key-value store. Each decision reached by an instance of the consensus algorithm (LastVoting) handles a batch of requests. The batch contains 32KB of data. We measure the amount of data that the system can process per second. The results are shown in Fig. 12a. 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 ReSync implementation of consensus with LibPaxos3 [?], etcd [?], Goolong [?], PSync [?], and Zab [?]. LibPaxos3 is a C implementation of multi-Paxos. The system processes requests which are simple arrays of bytes. We use requests size of 32KB to match the size of a batch in ReSync. etcd version 3.4.3, is a Go implementation of the Raft algorithm by [?]. We use the provided benchmarking tool with 1000 clients making request of 32KB. With more clients the throughput stops increasing and the system becomes unstable. Goolong is a multi-Paxos written in Go. It is also used in a key-value store and processes requests in batches of 100KB of data. PSync also implements the Last Voting algorithm but instead of a message accumulator it waits until it reaches the timeout. We also include the performance numbers from Zab [?] taken from the paper as a baseline reference.[3] Overall, ReSync follows the behavior exhibited by LibPaxos3 and Zab which are production systems used in industry. With a smaller number of replicas, we can see the cost of the round abstraction. With more replicas, this cost disappears as more messages help ReSync keep a tighter synchronization between processes. etcd and Goolong behave differently. etcd uses a compression layer which penalizes the system with few replicas. On the other hand, it is not strongly limited by the network. For 8 and 9 replicas it exceeds what the network bandwidth allows without compression. Goolong is CPU bound independently of the number of replicas[4].

*Byzantine consensus.* We have implemented in ReSync the normal decision algorithm from PBFT [?]. The algorithm 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. We compare our implementation against Bft-SMaRt [?]. Bft-SMaRt is a high-performance Byzantine fault-tolerant state machine replication library developed in Java. This library implements a protocol similar to PBFT together with complementary protocols to boost performance. The results are shown in Fig. 12b. Similar to the benign case, ReSync has a higher overhead for small numbers of replicas. The overhead tappers

[3] Benchmarking distributed algorithms fairly is difficult. Each system has many parameters which can be tuned to achieve the best performances. Therefore, we want to include a baseline comparison with a system tuned by its authors. To make the comparison fair, we run the test on old machines that match the evaluation of Zab performed in 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] The reported throughput is in line with the numbers reported in ?. We contacted on of the author to report this behavior.



(a) Benign test: ReSync against LibPaxos3, etcd, Goolong and Zab

(b) Byzantine test: ReSync against Bft-SMaRt

Fig. 12. Comparison for benign and Byzantine Consensus Algorithm

Table 2. Expressivity of ReSync versus PSync

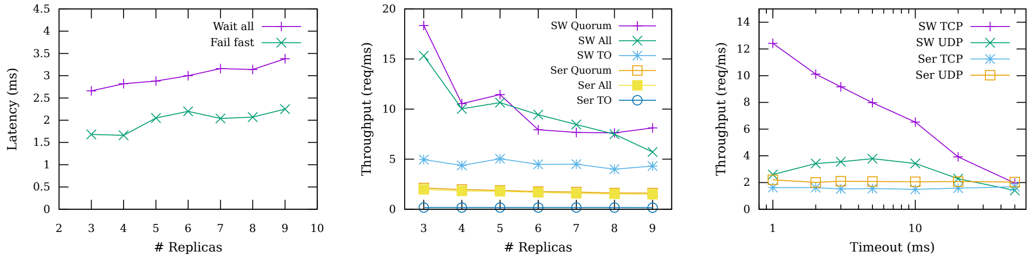
| Protocol | faults | PSync | ReSync | Message accumulator |
|---|---|---|---|---|
| LastVoting | benign | ✓ | ✓ | `Progress.timeout Progress.catchUp(true)` |
| One third rule | benign | ✓ | ✓ | `Progress.timeout Progress.catchUp(true)` |
| Two-phase commit | benign | ✗ | ✓ | `Progress.waitMessages Progress.catchUp(false)` |
| Failure detector | benign | ✗ | ✓ | `Progress.timeout Progress.catchUp(false)` |
| Θ-Model | benign | ✗ | ✓ | `Progress.timeout Progress.catchUp(false)` |
| Game of life | benign | ✗ | ✓ | `Progress.waitMessages Progress.catchUp(false)` |
| ViewChange | byzantine | ✗ | ✓ | `Progress.sync(F+1) Progress.catchUp(true)` |
| NormalOp | byzantine | ✗ | ✓ | `Progress.sync(F+1) Progress.catchUp(true)` |

off as the number of replicas increases. On the other hand, ReSync has a larger synchronization overhead due to the synchronization primitive that forces $F + 1$ processes to stay synchronized, which is more visible with a small number of processes. An interesting point to observe is the transition between 6 and 7 replicas for ReSync: 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 ReSync waits for the first 5 messages.

### 6.2 Comparison with PSync

We compared the expressiveness and performance of ReSync with PSync whose runtime we build on top of. Table 2 shows protocols that are implementable in ReSync and PSync. Last Voting by [?] (the round version of Paxos), one third rule by [?], and two-phase commit are consensus protocols. We implement the strong failure detector by [?](given in Appendix A.1.2) and the perfect failure detector in the Θ-Model by [?]. ViewChange (given in Appendix A.1.3) and NormalOp are protocols from the round-based version of PBFT by [?]. We highlight for each protocol the constructs used to implement the message accumulator. In PSync the programmer can vary only the timeout, which restricts the set of protocols implementable in PSync.

Fig. 12a includes a performance comparison between the implementation of Paxos in ReSync and the one in PSync. Although the core algorithm implemented in the two domain specific languages is the same—Last Voting [?]—the evaluation shows the advantage of using ReSync. Implementing a message accumulator gives an average speed up of 3.5×. We see that the performance curve of PSync is much flatter and the progress towards agreeing on a decision value is limited as the rounds end with a timeout. To get the maximal performance, we find the minimal timeout value



(a) Comparing progress conditions for the two-phase commit protocol with TCP and a 5ms timeout

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

(c) Effect of timeout values and transport layer in Paxos with 9 replicas progressing on quorum

Fig. 13. Comparing progress conditions with the Paxos consensus algorithm and the two-phase commit protocol

which allows the proposer to receive enough replies to form a quorum. With 3 replicas the timeout is 3ms, $4 - 7$ replicas use $4ms$, and $5ms$ for 9 replicas. With a message accumulator, ReSync does not need such tuning and can progress much faster.

There are no other domain specific languages that allow programming algorithms tolerant to both benign and byzantine faults. For the benign case we have experimented with DistAlgo [?]. However, their multi-Paxos implementation keeps the whole log of operations as a set in memory. The access gets slower as the algorithm progresses. Thus, we could not get performance measurement on longer runs as the algorithm stops making progress.

### 6.3 Comparing the effect of deployment conditions on message accumulators

We performed more tests on the implementation of our runtime by evaluating the performance of benign Paxos and two-phase commit using different message accumulators. In these 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 ReSync runtime. We report the speed of the system by the number of requests where each request is processed independently.

*Two-phase commit.* The two-phase commit protocol is interesting as the message accumulator inspects the received value. The protocol succeeds only if all the replica agrees. In Fig. 13a, we compare two message accumulator one which waits for all the replies before processing them in bulk and another one which terminates on the first negative reply. We can observe that the latency to process a request drops by one third when ReSync inspects the message content.

*Paxos.* We use Paxos in two different scenarios of deployment.
**Serializability** (Ser): The system calling our consensus implementation, ensures the serializability of decisions reached by different consensus instances. Only when one consensus instance has terminated the next consensus instance starts.
**Sliding Window** (SW): We use a sliding window to increase the system load. While a decision is still pending in a consensus instance, the system already starts the next consensus instance. The size of the sliding window is the limit on how many decision can be performed in parallel. Decisions can happen out-of-order but they are reordered before being applied to the system. We use a sliding window of size 20.

*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 progresses (switches rounds) either when it received messages from a quorum or timeout (Quorum), or when it received messages from all the replicas or timeout (All), or on timeout (TO). The results are shown in Fig. 13b.

We see that the (TO) approach, which coincides with the one used in PSync, 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 message accumulator: To switch rounds and make progress towards the agreement, 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 leader 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 for the duration of a message accumulator and implicitly for the duration of a round, which influences the synchronization of rounds across processes. Compared to other approaches where a timeout is used only to detect crashes, it may be better to *not* be conservative when picking the timeout in ReSync. ReSync's round synchronization primitives, `catchUp` and `sync`, are determined by the received messages, and timing out early results in several retries hence in sending more messages to synchronize the system.

Fig. 13c shows 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 ReSync timeout granularity is 1ms. It is interesting to observe that in the Ser scenario, the system is sightly faster over UDP than TCP. On the other hand, with a higher load TCP has a clear advantage. With higher load, we can observe that the performance starts degrading when the timeout gets below 5ms with UDP, while TCP makes a much better job at keeping the system synchronized with a small timeout. The TCP layer is highly-optimized, includes congestion control, and message retransmission. As TCP is lower down the networking stack, it can retransmit message more efficiently. At the application layer, the retransmission is more costly.

## 7 RELATED WORK AND CONCLUSION

In this section we discuss the relation with other close lines of work and future persepectives.

*Asynchronous programing languages.* Programming languages for distributed systems have been developed recently with the goal of formal guarantees: the language P [?] for asynchronous event-driven applications has a dedicated specification language and a testing tool for concurrency bugs. The language P simplifies the asynchronous control structure. It would be interesting in the future to combine the two approaches (ReSync and P) using for example P like structures to program the message accumulators.

Ivy [?] is a framework that uses a high-level language to specify and implement systems, and do verification. Protocols in Ivy are written in a general protocol language, using an event-based asynchronous semantics, out of which an EPR specification of the systems is extracted and executed under certain conditions. ReSync takes a different approach: it identifies synchronization primitives inherent to fault-tolerance. These programing concepts can be used by engineers who want to fine-tune their implementations or verification tools alike.

DistAlgo [?] is a domain specific language that focuses on synchronization primitives (and not on verification). It uses quantified queries to express synchronization conditions. DistAlgo mixes asynchronous programming and formal specifications as it uses quantified queries to express synchronization conditions. In contrast, ReSync proposes new synchronization primitives that alleviate the programmer from implementing systems using quantifiers.

There are also languages like Go and Erlang which are designed for programming in the distributed setting, or Rust which focusses on memory safety.

However, all these languages are 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 [?].

The reference in the review

*Round-based programing languages.* Synchronous round-based models [?], with their inherent logical time provided by the round number, relieve the programmer from implementing any notion of logical time. Partially synchronous round-based models [???] 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 [?] that we build upon. Psync [?] is a programming language based on a classic round based model, called Heard-Of [?]. Psync focuses on formal verification and execution of a strict subset of the partially-synchronous systems captured by the Heard-Of model. For example, two-phase commit is not implementable in PSync, although other consensus protocols are. The reason is that two-phase commit is designed for synchronous systems which requires that all messages sent in a round should be delivered in the round, which is not guaranteed by PSync. PSync guarantees this only eventually, that is, from some round on. The runtime of PSync is rigid and cannot be customized to accommodate protocol-specific requirements regarding the received messages. Contrary to PSync, or any other round model, ReSync proposes a new programming abstractions where round boundaries are under the control of the programer instead of a non-deterministic network or a restricted runtime. ReSync can handle byzantine faults, which is not the case of PSync, and the new synchronization primitives are essential to achieve this.

Similarly to PSync, virtual synchrony [?] gives the illusion of synchrony but does not allow custom optimization algorithms, and does not have synchronization primitives for byzantine protocols.

*Verification perspective.* Although ReSync has a round structure, which is known for simplifying the verification task, the focus of this paper is on overcoming the limitations of round-structures and the performance pitfalls which arise when verification is the primary goal of a domain specific language. Formal verification of ReSync programs is orthogonal and goes beyond the ambitions of this paper. However, the improvement of performance does not make the verification task more difficult. Rather, it allows a more structured two-step verification approach: (i) verify a round-based algorithm and (ii) verify the implementation of the round boundaries. Regarding (i), we would like to highlight that ReSync preserves the benefits of PSync, and more general of classic round structures, in proving safety properties for systems where network assumptions are required only to ensure liveness (like Paxos). These benefits are simpler proof arguments, i.e., invariants, due to a reduced number of interleavings compared with asynchronous computational models. Moreover, for a fragment of ReSync (closed rounds) existing verification techniques [??????] developed for round-based systems apply, and some of them also cover some byzantine protocols. However, we leave as future work a verification engine for ReSync, as it requires not only integrating known results but also, to address (ii), new reasoning rules that compose asynchronous and synchronous features. Specifying these combinations can be done by adapting the so called communication predicates in the Heard-Of model but the proof rules are future work.

*Conclusion.* ReSync is a new programming abstraction for fault-tolerant distributed protocols. It proposes the first round structure with parametrized round switching policies, making the performance of the round-based code adaptable to the networks the programs are written for. ReSync supports both benign and byzantine protocols and provides theoretical guarantees at compile time.

## ACKNOWLEDGMENTS

We thank the OOPSLA reviewers and Alexey Gotsman for useful comments and suggestions. Damien Zufferey is supported in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

more ack?

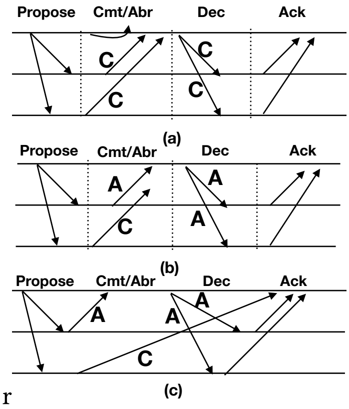


Fig. 14. Two-phase-commit: (a) and (b) are lockstep executions, (c) is a runtime execution.

# A APPENDIX

## A.1 Examples

*A.1.1 Two Phase Commit.* Fig. 5 presents the first two rounds — i.e., the first phase — of the protocol *two phase commit* in ReSync. 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 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 ??. Fig. 14(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. 14(b) shows an execution where the transaction is aborted. The first round, starts at line 2. 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 5) to all processes. Each process checks locally whether it can locally commit the transaction (without violating the consistency of the database) at line 16. In the second round, starting at line 2, processes send (at line 4) 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 15 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.

In ReSync, processes execute in lockstep a sequence of rounds called a `phase`. Each round has four methods: `send`, `receiveInit`, `receive`, and `finishRound`. Within a round, processes execute first `send`, which defines the messages to be sent, followed by `receiveInit` 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 `receiveInit` 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

```
// Initially no suspicion
var lastSeen: Map[ProcessID,Int]() = Map(for (i <- 0 until N) yield (i,0))
def getSuspected = lastSeen.filter{ case (_, last) => last > hysteresis }.keySet()

val rounds = phase(
    new EventRound[Set[ProcessID]]{
        def send: Map[ProcessID,Set[ProcessID]] = broadcast(getSuspected)
        def receiveInit = { Progress.timeout(period)}
        def receive(sender: ProcessID, suspected: Set[ProcessID]) = {
             Progress.unchanged }
        def finishRound(mbox: List[(ProcessID, Set[ProcessID])]) = {
            for ( (k,v) <- lastSeen ) { lastSeen(k) = v + 1 }
            for ((sender,suspected) <- mbox) {
                lastSeen(sender) = 0
                for (s <- suspected if lastSeen(s) != 0) {
                    lastSeen(s) = hysteresis + 1 } //suspect s
            }
            println("replica: "+id+" suspecting: " + getSuspected)
        }
    }
)
```

Fig. 15. Strong Eventually Fault Detector ReSync.

messages. Therefore, messages are received one by one until enough messages have been received and the method `finishRound` is called. The message accumulators of two-phase-commit is blocking if the network loses messages or processes crash. The call to `finishRound` receives as input the list of received messages (sorted in the arrival order), called mailbox or `mbox` for short.

ReSync 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 14(c) shows the runtime execution corresponding to the execution in Fig 14(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 lockstep executions of ReSync. From the client's perspective, the runtime execution in Fig 14(c) and the lockstep execution in Fig 14(b) are equivalent, the transaction is aborted in both.

*A.1.2 Fault Detector.* Failure detectors were introduced [?] 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 [?]. The program in Fig. 15 implements an eventually strong failure detector in ReSync. 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

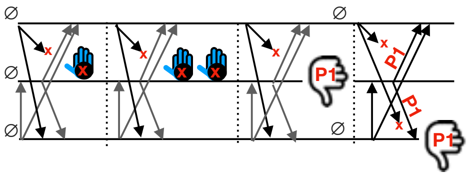


Fig. 16. An execution of the fault-detector in Fig. 15.

empty). In `finishRound` the process updates its set of suspected processes, based on the received messages in `mbox` for the current round.

Fig. 16 shows an execution of the protocol in Fig. 15, 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 11). In the third round $p2$ suspects $p1$ to be faulty in line 3, because `lastseen` of p2 is greater than $h$. Therefore, $p2$ broadcasts its suspicion about $p1$, which reaches $p3$. Consequently, $p3$ suspects $p1$ in line 15, since it did not receive any message from $p1$ in the current round.

The message accumulator in Fig 15 uses timeouts. Each processes waits for messages up to a timeout. The timeout semantics is defined in `receiveInit` by the instruction `Progress.timeout(period)` at line 8. 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 lockstep executions, 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, the algorithm implements a failure detector.

*A.1.3 ViewChange.* We give in Fig. 17 the implementation of the three rounds of ViewChange in ReSync.

```
val rounds = phase(
    new EventRound[Hist] {//Round: DoVC
       def send(): Map[ProcessID,Hist] = broadcast(val)
       def receiveInit(){
           Progress.timeout(to)
           Progress.catchUp(true)
           Progress.sync(F+1,true) }
       def receive(sender: ProcessID, payload: Hist) = {
           if (mbox.size >= 2*F+1) Progress.goAhead }
       def finishRound(box: List[(ProcessID, Hist)]) = {
           if(mbox.size > 2N/3)
               DoVC_mbox = mbox }
     },
     new EventRound[List[(ProcessID, Hist)]]{ //Round: Forward
       def send(): Map[ProcessID, List[(ProcessID, Hist)]] = Map( coord -> DoVC_mbox )
       def receiveInit(){
           Progress.timeout(to)
           Progress.catchUp(true)
           Progress.sync(false) }
       def receive(sender: ProcessID, payload: Hist) = {
           if (id == coord && mbox.size > 2N/3) Progress.goAhead }
       def finishRound(mbox:List[(ProcessID, List[(ProcessID, Hist)])]) = {
           if (id == coord && mbox.size > 2N/3) {
               decision = compute(mbox) } }
     },
     new EventRound[Either[Hist,List[(ProcessID, Hist)]]]{ //Round: NewView
       def send(): Map[ProcessID, Either[Hist,List[(ProcessID, Hist)]]] = {
          if(id == coord) broadcast(Left(decision))
          else broadcast(Right(DoVC_mbox)) }
       def receiveInit(){
           Progress.timeout(to)
           Progress.catchUp(true)
           Progress.sync(false)
       }
       def receive(sender: ProcessID, payload: Either[Hist,List[(ProcessID, Hist)]])= {
           if (mbox.size > 2N/3) Progress.goAhead
       }
       def finishRound(mbox:List[(ProcessID, Either[Hist,List[(ProcessID, Hist)]])]) =
            {
           if( mbox.size > 2N/3 && mbox.find(_.1 == coord ) )
               decision = check_computation(mbox)
               out(decision) }
    })
```

Fig. 17. View Change (PBFT) with authentication in ReSync.