PSync: a partially synchronous language for fault-tolerant distributed algorithms

Abstract

Fault-tolerant distributed algorithms are notoriously hard to implement correctly, due to asynchronous communication, uncertain message delays, and the occurrence of faults.

We introduce PSync a domain specific language which views asynchronous systems as synchronous ones with an adversarial environment that drops or delays messages. We have implemented a runtime system to support the execution of PSync programs over asynchronous networks. The advantage of PSync is twofold: the synchronous abstraction simplifies the design and implementation of fault-tolerant distributed algorithms and enables semi-automated formal verification. We have implemented an embedding of PSync in the SCALA programming language and evaluated our implementation on several consensus algorithms.

1. Introduction

Fault-tolerant distributed algorithms play a important role in many critical high-availability systems [Burrows 2006; Isard 2007; Hunt et al. 2010]. However, building such systems is challenging. Designers have to face: (1) asynchronous concurrency and (2) faults. Asynchrony means that processes can interleave in an arbitrary way and the network can choose to delay messages. Hardware faults can be transient, e.g., network partition, or permanent, e.g., crashes.

In a distributed system processes only have a limited view of the global state. However, in most distributed applications one has to enforce some global property. Ensuring such a property typically requires solving a consensus problem: roughly, each process has an initial value, and all non-faulty processes have to agree on a unique decision value from the set of initial values, even in the presence of faults and uncertainty in the timing of events. For instance, implementing a replicated state machine [Lamport 1998] can be reduced to using consensus iteratively.

From a theoretical perspective, a fundamental result on distributed algorithms [Fischer et al. 1985] shows that it is impossible to reach consensus in asynchronous systems where at least one process might crash. Consequently, a large number of algorithms have been developed [Dwork et al. 1988; Dolev et al. 1987; Chandra and Toueg 1996; Lamport 1998; Widder et al. 2012], each of them solving consensus under different assumptions on the type of faults, and the "degree of synchrony" of the system. These algorithms are typically given in natural language.

From a practical perspective, implementing fault-tolerant distributed algorithms is hard: network programming can be error-prone and algorithms are rarely implemented as theoretically defined, but modified to fit constraints and requirements of the system in which they are incorporated. Typically correctness arguments for these modifications are missing, leading to the execution of unproven protocols [Chandra et al. 2007]. There are currently no automated verification techniques that can be applied to these implementations.

In this paper, we propose a domain specific language PSync embedded in a general-purpose language for writing high-level implementations of distributed algorithms, that hide from the user the complexity of fault, asynchrony, and low-level network programming. We have implemented a runtime system that executes PSync programs over asynchronous networks, in the presence of faults. Moreover, we have designed a semi-automated verification engine to check that the high-level implementation meets its specification.

High-level computational model. Many distributed algorithms can be formulated in rounds [Lamport et al. 1982; Lynch 1996; Dwork et al. 1988; Charron-Bost and Schiper 2009; Keidar and Shraer 2006; Borran et al. 2012]. Conceptually, processes operate in lock-step, in a round they send messages, receive messages, and update their local state depending on the state at the beginning of the round and the received messages. Rounds are communication-closed [Elrad and Francez 1982] if all the messages sent in a round are received within the same round or discarded.

Our domain specific language is designed following the *heard-of* model (HO-model) [Charron-Bost and Schiper 2009], which is round based and communication closed. The central concept of this model are the *heard-of* sets HO, where the heard-of set of a process p contains the processes from which p may receive messages in a given round. The HO-sets are under the control of an adversarial environment, which simulates asynchrony, process crash, etc., by dropping messages. Thus, the network assumptions are given by *communication predicates*, which are linear temporal logic formulas (LTL) that impose global constrains on the heard-of sets in the computation. For instance, in a system consisting of n processes, the communication predicate $\Diamond\forall p.\, |\mathrm{HO}(p)| > n/2$ states that eventually, all processes may receive messages from a majority of the processes.

Domain specific language. A PSync program organizes the code into a sequence of rounds, and has a synchronous semantics where all processes execute the same code. The computation performed by a process in one round is encapsulated in two consecutive operations: a *send operation*, which is dedicated to sending messages, and an *update operation*, which defines updates on the local state based on the received messages. Each process has built-in variables, representing the number of processes in the network, the round number, and the HO sets. Before processes start executing the code of a round, the environment takes control over the program executions, modifying the values of the HO-sets of all processes. The LTL assumptions on the network defined by the communication predicates, restrict the environment's actions. These assumptions may differ between processes, and they don't impose constraints on data manipulated by the processes; they are encapsulated in a separate module.

We have implemented PSync as a shallow embedding in the SCALA programming language. In order to be executable, a PSync program is linked agains a runtime that manages the interface with the network and the resources used by the program. The code runs on top of an event-driven framework for asynchronous network applications, uses UDP to transmit data, in a context where processes may permanently crash. We assume the network satisfies the communication predicates. The main challenge in deriving an asynchronous implementation from a synchronous one is defining a procedure that decides the round switch while also allowing sufficiently many messages to be delivered. In general this decision depends on the communication predicate specified in the PSync program, and on the underlying network characteristics. We have focused on partially synchronous architectures [Dwork et al. 1988], where bounds on message delays and the process computation speeds exist but are not known a priori. In this case the round switch can be implemented using increasing timeouts.

Semi-automated verification of PSync programs From the verification perspective, distributed algorithms are a very challenging class of systems because of several sources of unboundedness: messages come from unbounded domains, the number of processes is a parameter, and executions have an unbounded number of interleavings. Indeed, most of the verification problems are undecidable for parameterized systems [Apt and Kozen 1986]. However, the structure of PSync together with its synchronous semantics make it a prime candidate for automated verification.

We have developed verification techniques for PSync programs based on global invariant checking, therefore automating the Hoare style of reasoning. To this, properties of program states or properties of relations between them are given in the assertion logic $\mathbb{CL}$ [Dragoi et al. 2014]. The specification of a PSync program and the communication predicates are given in LTL over atomic propositions in $\mathbb{CL}$.

To verify that a PSync program meets its specification, we assume the user provides inductive invariant candidates, written in the assertion logic, and the verification engine automatically checks their validity and the fact that they imply the specification assuming the communication predicates. To discard the verification conditions we have used a semi-decision procedure for checking the validity of implications between $\mathbb{CL}$ formulas [Dragoi et al. 2014].

Contributions. In this paper, we present a domain specific language PSync for writing high-level implementations of distributed algorithms. The interplay between asynchrony and faults is at the heart of any distributed algorithm and our domain specific language addresses this problem by seeing any asynchronous system as a synchronous one where the environment is an adversary that drops or delays messages. This shift to a synchronous point of view simplifies both developing algorithms, and their correctness proofs, without restricting the computational power.

We have implemented a runtime for PSync that executes over an asynchronous network. The runtime design ensures that from the point of view of a process, the asynchronous executions are indistinguishable from the synchronous ones. A PSync program can be interfaced with other applications through asynchronous callback methods. In our experimental evaluation we have focused on consensus algorithms solving consensus under different assumptions with an high-level implementation in PSync. These implementations tolerate transient network partitions and crash-faults. We have used these high-level PSync programs to implement a distributed key-value store, a distributed lock manager, and a protocol for dynamic view change.

Moreover, the correctness of a PSync program can be semi-automatically verified, producing a proof of the high-level implementation. Therefore, provided that the implementation of the runtime is correct, proving that a PSync program solves problems like consensus implies that the low-level implementation solves consensus as well.

Sec 2 defines the domain specific language PSync, while Sec. 3 gives a procedure to execute a PSync program over an asynchronous code. The verification techniques are presented in Sec. 4. Finally Sec. 5 presents an embedding of PSync in the SCALA languages and Sec. 6 presents the experimental evaluation of the embedding.

2. PSync syntax and semantics

PSync is a domain specific language embedded within a general program language, that offers specialized syntactic constructs for programming distributed algorithms.

2.1 Syntax

The core of a program in PSync consists of the variable declaration, a set of function definitions, a non-empty list of *round modules*, and a *communication predicates module*. The specification of the problem to be solved by the program is given in the *specification module*. To prove the program correct w.r.t. the given specification the verification engine relies on a set of user defined annotations, given in the *properties module*.

All processes run the same code, i.e., the sequences of rounds defined in the PSync program, starting with an entry procedure to initialize their local variables. The **ENTRY** procedure has input parameters which can be either data values or callback functions, defined in the host programming language. In each round a process goes though three stages. First all processes are under the control of an environment which decides the faults of the round. Then processes send messages either to designated processes in the network, such as task leaders, or to the entire network. Finally processes update their local state based on the received messages. Therefore each round module defines two operations: **SEND** and **UPDATE**, implementing the two stages of a process that are under its control.

Each PSync program declares *process variables*, which represent data values such as integers, or any other type supported by the host language. Moreover, every program has four built-in variables: N, an integer representing the number of processes in the network, rnd, an integer representing the round number, HO of type Set <**Id**>, a set of process identities representing the heard-of set, where **Id** is a type representing process identities, and Mbox of type **Id** $\times$ Data representing the set of received messages in the current round, where each message contains the sender's identity and the payload.

Example 2.1. Fig. 1 shows the PSync program implementing the *One Third Rule* [Charron-Bost and Schiper 2009] (*OTR*) algorithm for solving consensus. is an encoding of [Brasileiro et al. 2001] in the HO-model.

```
VARIABLE
  x, dec, in : int
ENTRY (int v)
  x := v; in := v;

ROUND R
  SEND
    send(x,all);
  UPDATE
    Occ: Set<Id>
    if (Mbox.size > 2N/3)  x:=mmor(Mbox);
    Occ = occurences(x,Mbox);
    if (Occ.size > 2N/3)  dec := x;
  AUX_FUN
    int mmor(Set<Id×int> S) {...}
    @ensures $result = v \wedge Q = \{p \mid (v,p) \in S\} \wedge$
    $\forall q.\, Q(q) = \{t \mid (u,t) \in S \wedge (u,q) \in S\} \wedge$
    $(|Q(q) < |Q| \vee (|Q(q)| = |Q| \wedge v \leq u)$
    Set<Id> occurences(int v, Set<Id×int> S) { ... }
    @ensures $result = Q \wedge Q = \{p \mid (v,p) \in S\}$
PRED com_pred
assume $\Diamond(\exists K\, \forall p.\, \mathsf{HO}(p) = K \wedge |K| > 2N/3 \wedge$
          $\Diamond(\forall p.\, |\mathsf{HO}(p)| > 2N/3))$
SPEC
A := $\Box\, \forall p, q.\, \mathsf{dec}(p) \neq ? \wedge \mathsf{dec}(q) \neq ? \Rightarrow \mathsf{dec}(p) = \mathsf{dec}(q)$
V := $\Box\, \forall p \exists q.\, \mathsf{dec}(p) \neq ? \Rightarrow \mathsf{dec}(p) = \mathtt{in}(q)$
I := $\Box\, \forall p.\, \mathsf{dec}(p) \neq ? \Rightarrow \mathsf{dec}(p) = \mathsf{dec}'(p)$
T:= $\Diamond\, \forall p.\, \mathsf{dec}(p) \neq ?$
spec := [A,V, I, T];
PROP
$Inv := \exists q.\, \big(M = \{t \mid x(t) = \mathtt{in}(q)\} \wedge |M| > 2N/3 \wedge$
          $\forall p.\, \mathsf{dec}(p) \neq ? \Rightarrow \mathsf{dec}(p) = \mathtt{in}(q)\big)$
$Inv_1 := \exists q\, \forall p.\, x(p) = \mathtt{in}(q)$
$Inv_2 := \exists q\, \forall p.\, \mathsf{dec}(p) = \mathtt{in}(q)$
Invariants := $[Inv, true, Inv_1, Inv_2]$;
```

Figure 1. The One Third Rule algorithm in PSync

Each process has three local variables x, dec, and `in` of type integer. The variables x, resp. `in`, encode the current value that a process proposes and the dec variable eventually received the decision value. After variable that are not initialized in **ENTRY** receive a default '?' value.

Processes start by running the entry procedure, which initializes the variables x and `in`. Then they repeatedly run the send operation and the update operation of the round R. Given a process p, Mbox denotes the set of received messages, a set of tuples of the form (q, v), where q is the identity of the sender and v is the message payload. In this case the payload is q's x value. The Mbox of p is restricted to contains only messages form q if $q \in \mathsf{HO}(p)$.

During the update, a process p checks if it received more than $2N/3$ processes. If this is the case, p updates its local variable `x` with the minimal most often received value (mmor). If the condition does not hold, the value of `x` stays unchanged. As the HO-set differs between processes, it can be that only some processes update `x`. In the second `if` statement, a process p dec on x if it received the same value from more than $2N/3$ processes. Therefore, the algorithm establishes a two third majority of process agreeing before taking a decision. To prove termination of the algorithm, the communication predicate guarantees two good rounds where processes receive from more than $2N/3$ processes. The first good round establishes the majority, and the second round witnesses that majority.

Environment transition. The first step of a round execution is under the control of the environment, which synchronously decides the faults that occur in that round, by assigning non-deterministic values to the HO-sets. The environment determines the crashed processes by not including their identities in any HO-set.

Round operations. The statements of a *send operation* **SEND** are defined by send_stmt in Fig. 2.1. This operation must contain at least one send statement and it cannot access the variables HO or Mbox. Moreover, none of the process variables are updated, however, to compute the payload of a message external pure functions [1] can be used. There are algorithms which require some level of coordination, e.g., designated task leaders. In order to use a point-to-point communication with designated processes, the auxiliary functions should implement ways of defining their identities, using only the built-in process variables (and any other auxiliary variable visible only within the function's scope).

The statements of an *update operation* **UPDATE** are given by upd_stmt in Fig. 2.1. The syntax of the update operation is restricted to a guarded command-like language [Dijkstra 1975]. However processes can perform more involved computations using external pure functions.

External functions. Besides the two operations, a round module contains a list of auxiliary pure functions, defined in the host language, introduced by the keyword **AUX_FUN**. They can be called by any operation within the scope of the module where they are declared. These are sequential functions and their syntax and semantics is that of the host language. We assume that these functions terminate within a bounded number of steps where the bound depends on the number of processes N, and on the input parameters of the entry procedure.

Variables and types. A PSync program can manipulate data of various types: the language supports basic types like int, bool, and float, tuples of basic types, and container types, e.g., Set <int>, Set <int × int>. We assume that basic types come with the usual operations, e.g., addition, multiplication. In particular, we assume that the Set <T> type is enhanced with methods size, that returns the cardinal of the set received as input, contains for testing set membership, union to compute the union of two sets, etc.

PSync annotations. The assumptions in the **PREDS** module and the properties in the **SPEC** module are defined in LTL with atomic formulas in the assertion logic $\mathbb{CL}$. The logic $\mathbb{CL}$ can express state properties of algorithms in the heard-of model, and also relations between states, so is a natural assertion language for PSync. The **PREDS** module contains a list of formulas called communication predicates, defining the assumptions on the network: these formulas refer only to the HO variables, the round number, rnd, and function symbols interpreted over process identities. The **SPEC** module constrains only the process variables declared in the program. The PROP module contains a list of auxiliary properties expresses in the assertion logic $\mathbb{CL}$ which are used by the verification engine to prove the program's correctness. These properties include the summary of the auxiliary functions, i.e., the relation between the input and the output of the function. Sec.4 contains details on $\mathbb{CL}$.

[1] Pure functions have no side-effects. Their result is deterministic and it depends on their inputs.

```
     program  ::=  var_decl; entry; rounds;
                   com_preds; spec
    var_decl  ::=  ident: type var_decl
       entry  ::=  ENTRY (var_decl) stmt
      rounds  ::=  ROUND  round_name
                   SEND  var_decl; begin; send_stmt; end;
                   UPDATE  var_decl; begin; upd_stmt; end;
                   PROPS  list_of_formulas
                   AUX_FUN  fun_def
                |  rounds; rounds
   com_preds  ::=  PREDS  preds_name
                   assume_stmt;
        spec  ::=  SPEC assert_stmt;
    env_stmt  ::=  skip | assign | if_then_else
                |  env_stmt ; env_stmt
   send_stmt  ::=  send(msg, dest)
                |  assign | if_then_else
                |  send_stmt; send_stmt
    upd_stmt  ::=  assign | if_then_else
                |  upd_stmt; upd_stmt
 assume_stmt  ::=  assume formula
                |  assume_stmt; assume_stmt
 assert_stmt  ::=  assert formula
                |  assert_stmt; assert_stmt
if_then_else  ::=  if expr then stmt else stmt
      assign  ::=  x:= expr
         msg  ::=  (x_1, ..., x_n), n ≥ 1
        dest  ::=  Id _expr | all
```

Figure 2. PSync syntax.

2.2 Semantics

A state of the program is a partial mapping $S : \mathbf{Id} \rightharpoonup \mathbf{L}$ from process id's to process local states. A process state is a tuple (σ, stmt), where:

- σ is a valuation for the process's local variables, and
- stmt is a sequence of instructions to be executed.

Besides the local variables declared in the program, and the built-in ones, each process p is enhanced with a variable OutBox representing a buffer of messages that stores the messages in transit sent by p in the current round. These variables are reset by each environment transition, because messages not received in one round are dropped. An initial configuration of the program is defined by a mapping $S_0 : \mathbf{Id} \rightharpoonup \mathbf{L}$ such that for every p in the domain of S_0, $S_0(p) = (\sigma_0, \mathsf{stmt})$ where $\sigma_0(\mathsf{rnd}) = 0$, $\sigma_0(\mathsf{OutBox}) = \sigma_0(\mathsf{HO}) = \emptyset$, $\sigma_0(\mathsf{N}) = |dom(S_0)|$, and stmt is the sequence of rounds defined in the program. All the other process variables are

$$\frac{\langle \sigma, \mathbf{ENTRY}(\vec{p}); \mathbf{ROUND}\ \mathsf{R}_1 \dots; \mathbf{ROUND}\ \mathsf{R}_n \dots, \rangle}{\langle \sigma, \mathbf{ENTRY}; \mathbf{OP}(\mathsf{R}_1); \dots; \mathbf{OP}(\mathsf{R}_n); \mathbf{ROUND}\ \mathsf{R}_1 \dots \mathbf{ROUND}\ \mathsf{R}_n \dots, \emptyset \rangle}\ \mathsf{Init\_rounds}$$

$$\frac{\langle \sigma, \mathbf{send}(\mathsf{msg}, \mathsf{expr}); \mathsf{stmt} \rangle}{\langle \sigma, \mathsf{stmt}, \mathsf{OutBox} \cup \{(\sigma(id), \sigma(\mathsf{expr}), \sigma \circ \mathsf{tag}(\mathsf{msg}))\} \rangle}\ \mathsf{Send\_msg}$$

$$\frac{\langle \sigma, \mathbf{send}(\mathsf{msg}, \mathbf{all}); \mathsf{stmt} \rangle}{\langle \sigma, \mathsf{stmt}, \mathsf{OutBox} \cup \{(\sigma(id), r, \sigma \circ \mathsf{tag}(\mathsf{msg})) | r \in \Pi\} \rangle}\ \mathsf{Send\_all}$$

$$\frac{\langle \sigma, \mathbf{UPDATE}; \mathsf{stmt} \rangle}{\langle \sigma[\mathsf{Mbox} \leftarrow \{(s, \sigma(id), m) \in \mathsf{OutBox}(s) | s \in \sigma(\mathsf{HO})\}], \mathsf{stmt} \rangle}\ \mathsf{Set\_Mbox}$$

$$\frac{\langle \sigma, \mathbf{ENV}; \mathsf{stmt} \rangle}{\langle \sigma[\mathsf{HO} \leftarrow *, \mathsf{OutBox} \leftarrow \emptyset], \mathsf{stmt} \rangle}\ \mathsf{Env} \qquad \frac{\langle \sigma; \mathbf{SEND}; \mathsf{stmt} \rangle}{\langle \sigma, \mathsf{stmt} \rangle}\ \mathsf{Send}$$

$$\frac{\langle \sigma, \mathsf{end}; \mathbf{ENV}; \mathsf{stmt} \rangle}{\langle \sigma[\mathsf{rnd} \leftarrow \mathsf{rnd} + 1, \mathsf{Mbox} \leftarrow \emptyset], \mathsf{stmt} \rangle}\ \mathsf{Inc\_round}$$

Figure 3. Operational semantics for internal process steps.

$$\frac{\begin{array}{c}\langle S \in [\mathbf{Id} \to \mathbf{L}] \rangle \quad S[id] = (\sigma, \mathsf{s}; \mathsf{stmt}),\ \text{with} \\ \mathsf{s} \notin \{\mathbf{ENV}, \mathbf{SEND}, \mathbf{UPDATE}\}\end{array}}{\begin{array}{c}\langle S' \in [\mathbf{Id} \to \mathbf{L}] \rangle \quad S'|_{\mathbf{Id} \setminus id} = S|_{\mathbf{Id} \setminus id} \\ S'[id] = (\sigma', \mathsf{stmt})\ \text{and}\ \langle S[id] \rangle \xrightarrow{\mathsf{s}} \langle S'[id] \rangle\end{array}}\ \mathsf{local}$$

$$\frac{\begin{array}{c}\langle S \in [\mathbf{Id} \to \mathbf{L}] \rangle \quad \forall id \in \mathbf{Id},\ S[id] = (\sigma^{id}, \mathsf{s}; \mathsf{stmt}^{\mathsf{id}}), \\ \mathsf{s} \in \{\mathbf{ENV}, \mathbf{SEND}, \mathbf{UPDATE}\}\end{array}}{\begin{array}{c}\langle S' \in [\mathbf{Id} \to \mathbf{L}] \rangle \\ \forall id \in \mathbf{Id},\ S'[id] = (\sigma^{id}, \mathsf{stmt}^{\mathsf{id}})\end{array}}\ \mathsf{sync}$$

Figure 4. Parallel small-step semantics

uninitialized in S_0, i.e., for every variable x, $\sigma_0(\mathsf{x}) = ?$, where ? is a special constant included in any type.

The operational semantics of PSync is defined first defined for a single process and then, extended to multiple processes using parallel composition.

The semantics of the actions executed by one process are defined in Figure 3 (for brevity, we omit the transition rules for assignments, if_then_else statements, etc). The first rule in Figure 3 states the fact that the rounds declared in a PSync program are supposed to be executed infinitely often by every process, in the order in which they are declared. Given R a round module, **OP**(R) denotes the sequence of operations in its declaration preceded by an environment transition denoted **ENV**:

$$\mathbf{OP}(\mathsf{R}) := \mathbf{ENV}\ \ \mathbf{SEND}\ \mathsf{send\_stmt}\ \ \mathbf{UPDATE}\ \mathsf{upd\_stmt}$$

We first define a set of *small-step runs*, $\mathit{SmallRuns}(A)$, based on small-step semantics for the transitions of a PSync program given Figure 4. The small-step runs are defined by the parallel composition of the transition rules of each process where the environment transition is executed synchronously and processes synchronize at the beginning of each operation. Note that actions of different process between two synchronization steps can be interleaved in any order because there is no dependency between them, i.e, the transitions from Fig. 3 read and write only the local state of a process.

We define the set of *lockstep runs*, based on a lockstep semantics of the system of transitions in Figure 4, where each process executes atomically all local transitions between two synchronization steps. An environment lockstep is an environment transition, and a send, respectively update lockstep, is the execution of the synchronous transition associated with **SEND**, respectively **UPDATE**, composed with the atomic execution of all the local transitions defined by the statements of the corresponding round operation.

Formally, the set of small-step runs of a PSync program A, i.e., $\mathit{SmallRuns}(A)$, is defined by sequences of runs of the form $S_0 S_1 S_2 \dots$, where S_i are program states such that $S_i \xrightarrow{\mathsf{sync/local}} S_{i+1}$ for all $i \geq 0$ and S_0 is the initial configuration.

The set of lockstep runs is $\mathit{LockRuns}(A) = \{e \downarrow_{\mathsf{sync}} | \ e \in \mathit{SmallRuns}(A)\}$, where $e \downarrow_{\mathsf{sync}}$ is the projection of the execution e on the global states reached after a synchronization step (defined by the sync transition rule in Fig. 4).

In the following we give more details concerning the semantics of the environment and of the actions executed by processes in each of the two round operations.

Send operation. The code of **SEND** defines the messages sent by a process in a round. The semantics of the send statements is defined by the rules send_all and send_p. When process p sends a message msg to process q, the tuple $(p, q, \sigma \circ \mathsf{tag}(\mathsf{msg})$ is added to its buffer OutBox of messages in transit, where $\sigma \circ \mathsf{tag}(\mathsf{msg})$ is the message payload tagged with the round number in which it is send.

Update operation. The update operation modifies the values of the declared process variables, based on the process's local state at the end of the send operation and the set of received messages. Therefore, the first step of the update operation, i.e., the one is executed synchronously by all processes, defines the value of Mbox: it consists of all tuples (q, msg) where $q \in \mathsf{HO}(p)$ and (q, p, msg) belongs to the buffer of messages in transit of process q and the message msg is tagged by the current round number. The last statement of the update operation increments the round number and resets the content of mailboxes (see rule Inc_rounds).

Predicates module. The communication predicates **PREDS** module make assumptions on to the level of synchrony between the processes and the number of faults. They are interpreted over lockstep runs of the program. The temporal operators have the usual interpretation, i.e., $\Diamond\varphi$ states that the computation reaches a state satisfying φ, and $\Box\varphi$ states that all reachable states satisfy φ.

Roughly, a communication predicate states that every run goes through several "good rounds" when the HO-sets are populated enough to ensure progress. The progress can be quantified in terms of the number of variables updates performed by the program (in the update operation). Using constraints on the HO-sets one can specify various conditions on the crash faults and the omission faults in the system [Charron-Bost and Schiper 2009].

Set of computations defined by a program. The set of lockstep runs of a PSync program is defined independently of the network on which the program is run.

Definition 2.1 (PSync semantics). We define the semantics of a PSync program by the set of lockstep runs that satisfy the communication predicates. Formally, the set of runs of a program A in PSync is

$$\begin{aligned}\mathsf{Runs}(A) \quad = \quad & \{\rho = S_1 S_2 \ldots \in \mathit{LockRuns}(A) \mid \\ & \rho \models \psi, \forall \psi \in \mathsf{PRED}\},\end{aligned}$$

where $\models$ is the usual satisfaction relation of LTL.

3. From PSync to asynchronous code

In this section we define the procedure for the asynchronous execusion of a PSync program A, denoted $\tilde{A}$.

Network assumptions. The implementation $\tilde{A}$ is designed for partially synchronous networks [Dwork et al. 1988], where messages are exchanged asynchronously, failures are non-byzantine, i.e., messages can only be delayed indefinitely but they cannot be altered, and crashed processes do not recover. Moreover, we assume that the network satisfies the constraints imposed by the communication predicates.

A network is partially synchrony if there is a bound Δ on the message delivery time, and correct processes have a bound ϕ on the rate a processor can execute faster than another, but their values are unknown. If the network were synchronous, then Δ and ϕ would be fixed, and one could precisely estimate the time a process needs to perform a round. However, when Δ and ϕ are unknown we take the classic approach, that iterates over all their possible values.

Safety communication predicates. The distributed algorithms we have considered are safe no matter in which environment they are executed, so the communication predicates impose only liveness constraints. However, if the PSync program is safe only under certain network assumptions, then the asynchronous implementation $\tilde{A}$ relies on a user defined implementation, that allows processes to check locally weather the safety communication predicates are satisfied. In the following we consider PSync programs that require only liveness communication predicates.

Implementation skeleton. Figure 3 shows the control flow of the asynchronous code obtained from a PSync program.

Any execution starts with a process initialization phase, denoted **ENTRY**. The boxes **ENTRY**, **SEND** and **UPDATE** represent functions interpreting in the host language the entry procedure and the round operations with the same names

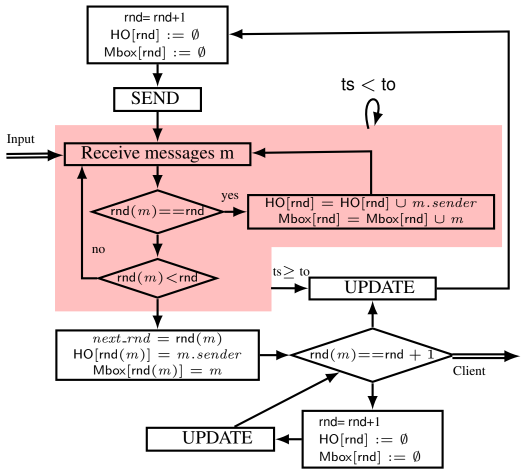

Figure 5. PSync implementation process control flow

from the PSync program. The interpretation of the send operation uses macros to define an implementation for point-to-point and broadcast communication in the host language. We assume that these functions always terminate, in a number of steps bounded by the number of processes and the entry values. The entry procedure and the update operation are written in the host programming language.

In each round processes start by sending messages and continue with receiving messages. Contrary to the PSync semantics, the mailbox, Mbox, and the HO-sets [2], are determined by the asynchronously received messages. We denote by $\mathsf{HO}[\mathsf{rnd}]$, respectively $\mathsf{Mbox}[\mathsf{rnd}]$, their values in the current round (initially they are empty). When a process receives a message tagged by its current round number, rnd, it adds the payload of the message and the identity of the sender to $\mathsf{Mbox}[\mathsf{rnd}]$, and the sender's identity to $\mathsf{HO}[\mathsf{rnd}]$.

Because of asynchrony a process may receive a message tagged by a round number bigger than its current round rnd. In this case it speeds up the computation until it reaches the round of the received message. For all the intermediary rounds it uses trivial values for the Mbox and HO. Messages tagged by past rounds are dropped.

The time span of a round is bounded by a *timeout* value, i.e., a process is allowed to receive messages and update the mailbox until the *timeout* expires. Time passage is modeled by a distinguished process variable denoted ts, which is incremented up to the value of an integer variable to defining the current estimation of Δ and ϕ. For simplicity we underline the time passing only in the block implementing the message reception within one round, because in this block the timeout determines how long a process waits before moving to the next round. We assume a bounded clock drift between processes, which cannot be directly observed

[2] When the PSync program uses point-to-point communication, to ensure that the values of the HO-sets in the implementation match the ones in the semantics, we assume additional dummy messages sent to every process. Those messages are ignored by the mailbox.

by processes. Once enough messages are received or the timeout is reached the control passes to **UPDATE** [3].

3.1 Implementation semantics

A state of a PSync implementation is a map $\tilde{S} : \mathbf{Id} \to \tilde{\mathbf{L}}$ from process id's to process local states together with the set of messages which are in transit, denoted Pool. The elements of Pool are tuples (s, d, m), where s and d are the sender, respectively the destination of the message, and m is the message payload tagged by the round number when it was sent. A process local state is a tuple $(\varepsilon, \mathsf{stmt})$, where ε is an evaluation for the process local variables, and stmt is the sequence of statements to be executed by the process.

The initial state is defined by an empty $\tilde{\mathsf{Pool}}$ and a mapping $\tilde{S_0} : \mathbf{Id} \to \tilde{\mathbf{L}}$, such that for every process p, $\tilde{S_0}(p) = (\varepsilon_0, \mathsf{stmt})$ where $\varepsilon_0(\mathsf{rnd}) = 0$, $\varepsilon_0(\mathsf{N}) = |dom(S_0)|$, and stmt is the sequence of statements defined by the transformation described above.

Let $AsyncExec(A)$ be the set of asynchronous executions obtained by taking the asynchronous parallel composition of processes local transition systems. For each process, its local transition system is defined by the control-flow graph in Fig. 3. If a process p is crashed in a state $\tilde{S}$ then $\tilde{S}[p] = \emptyset$. We consider only permanent crashes, so in any execution if $\tilde{S}[p] = \emptyset$ then in any subsequent state $\tilde{S'}$, $\tilde{S'}[p] = \emptyset$.

We define the set of executions of the implementation $\tilde{A}$, denoted $\mathsf{Exec}(\tilde{A})$, by restricting the executions in $AsyncExec(A)$ to those that satisfy the communication predicates. Next we define the interpretation of the communication predicates over asynchronous executions.

Interpreting communication predicates Let π be an execution in $AsyncExec(\tilde{A})$. We define the sequence $Receives(\pi)$ of tuples of local states reached at the beginning of the **UPDATE** block, ordered by round number. Each tuple contains one local state for each process. Formally, let $\tilde{S}[p]_0, \tilde{S}[p]_1 \ldots$ be the sequence of local where $\tilde{S}[p]_i$ is the state reached by process p at the beginning of the **UPDATE** block in round i. If a processes p crashes in π then we consider $\tilde{S}[p]_i = \emptyset$ for all rounds i after the crash. We define $Receives(\pi)$ as the sequences $U_1 U_2 \ldots$ where $U_i[p] = \tilde{S}[p]_i, \forall i \geq 1, \forall p \in \mathbf{Id}$. Let ψ a communication predicate in LTL with atomic formulas in the assertion logic, then

$$\pi \text{ satisfies } \psi \text{ iff } Receives(\pi) \models_{\text{LTL}} \psi.$$

The temporal operators of ψ are interpreted over $Receives(\pi)$ as usual, while the atomic propositions are interpreted over non-crashed processes.

Definition 3.1. A state $\tilde{S} \in [\mathbf{Id} \rightharpoonup (\tilde{\mathbf{L}} \cup \emptyset)]$ satisfies a formula φ in the assertion logic iff $\tilde{S} \downarrow_{\neg\emptyset} \models \varphi$, where $\models$ is the satisfaction relation of the assertion logic, $\tilde{S} \downarrow_{\neg\emptyset}$ is the projection over non-crashed processes in $\tilde{S}$(processes p such that $\tilde{S}[p] \neq \emptyset$).

[3] If the PSync program includes safety communication predicates, timeouts are overruled by the non-satisfaction of the safety predicate.

Definition 3.2 (Implementation semantics). The set of executions $\mathsf{Exec}(\tilde{A})$ of the implementation $\tilde{A}$ consists of all the executions $\pi \in AsyncExec(\tilde{A})$ such that π satisfies ψ for all LTL formulas ψ in the predicate module of A.

3.2 Indistinguishability relation

Let $\overline{\mathsf{Runs}}(A)$ be the set of execution in $SmallRuns(A)$ of a PSync program A that respect the communication predicates under the small-step semantics.

In this section we show that for any execution in $\mathsf{Exec}(\tilde{A})$ there is an indistinguishable execution in $\overline{\mathsf{Runs}}(A)$, from the point of view of any process p. Indistinguishability is proved by building for any asynchronous execution π in $\mathsf{Exec}(\tilde{A})$ a stuttering equivalent [**?**] execution ρ in $\overline{\mathsf{Runs}}(A)$, which preserves only the values of variables declared in the PSync program. The asynchronous code uses more variables than the PSync program. However, we consider two executions π and ρ indistinguishable if processes observe the same values for the variables of the PSync program. Exceptionally, the built-in variables HO and Mbox are allowed to have different values outside the **UPDATE** operation: in the asynchronous executions π, Mbox and HO are computed though several iterations of message reception while in the PSync run ρ they are synchronously modified by the environment. Since Mbox and HO are accessed only by update operations one can show that for any asynchronous execution there is a run of the PSync program such that any process sees the same values for the PSync program variables in the two executions. In the following we formalize the state equivalence and the stuttering relation that the indistinguishability relation is based on.

Definition 3.3 (State equivalence). Let $PVars$ be set of process variables of a PSync program. Two states $s = (f, \mathsf{s}; \mathsf{stms})$ and $s' = (f', \mathsf{s}'; \mathsf{stmt}')$ in $\mathbf{L}$ and respectively $\tilde{\mathbf{L}}$ are equivalent denoted $s \equiv s'$ iff (1) $f|_{PVars \setminus \{\mathsf{HO},\mathsf{Mbox}\}} = f'|_{PVars \setminus \{\mathsf{HO},\mathsf{Mbox}\}}$, $f'(\mathsf{HO}) \subseteq f(\mathsf{HO})$ and (2) if s and s' are statements of **UPDATE** then $f|_{\{\mathsf{HO},\mathsf{Mbox}\}} = f'|_{\{\mathsf{HO},\mathsf{Mbox}\}}$.

Definition 3.4 (Stuttering equivalent local executions). Let p be a process and θ, $\tilde{\theta}$ two sequences of local states of p from $\mathbf{L}$ and $\tilde{\mathbf{L}}$, respectively. Then, θ and $\tilde{\theta}$ are stuttering equivalent, denoted by $\theta \simeq \tilde{\theta}$ iff $\theta = \theta_0\theta_1 \ldots$ and $\tilde{\theta} = \tilde{\theta_0}\tilde{\theta_1} \ldots$ such that

- each θ_i, $\tilde{\theta_i}$ is non-empty and finite, for all $i \in \mathbb{N}$ and
- for all $s \in \theta_i$ and $s' \in \tilde{\theta_i}$, $s \equiv s'$, for all $i \in \mathbb{N}$, or for all $i < k$ if there exists k such that θ_k, $\theta_{k+1}, \ldots$ are sequences of states where p is crashed.

Definition 3.5 (Indistinguishable executions). Let p be a process, A an algorithm, $\rho \in \overline{\mathsf{Runs}}(A)$ a run of A and $\pi \in \mathsf{Exec}(\tilde{A})$ an execution of the implementation of A. The executions ρ and π are *indistinguishable from the point of view of a process* p iff $prj(\pi, p) \simeq prj(\rho, p)$ where $prj(\pi, p)$, $prj(\rho, p)$ are the sequences of local states reached by p in the execution π, respectively ρ, and $\simeq$ is the stuttering equivalence. We call two executions indistinguishable if they are indistinguishable from the point of view of every process, except for processes that crash, whose executions have to be indistinguishable up to the states where the process crashes.

Theorem 3.1. Given a PSync program A, for every execution $\pi \in \mathsf{Exec}(\tilde{A})$ of the implementation of $\tilde{A}$, there exists a run $\rho \in \overline{\mathsf{Runs}}(A)$ of A such that π and ρ are indistinguishable.

4. Verification of PSync

The specification of a PSync program is defines by LTL formulas over the assertion logic $\mathbb{CL}$ [Dragoi et al. 2014].

The variables of $\mathbb{CL}$ are interpreted over the different types declared in the program. The value of the program variable x of type T of a process p is denoted in the logic by the term $\mathsf{x}(p)$, where x is a function symbol of type $\mathsf{Id} \to \mathsf{T}$. In order to characterize global configurations, $\mathbb{CL}$ uses universal quantification over variables of type Id, set comprehensions, and cardinality constraints.

The specification of a PSync program that solves consensus, and other agreement problems like k-set agreement, or lattice agreement, are expressible in LTL over $\mathbb{CL}$: all four properties defined by the specification of *OTR* in Figure 1, are expressed using $\mathbb{CL}$ where dec and in are function symbols associated with the program variables with the same names. Irrevocability is an invariant for the transitions of the program: it defines the relation between the values of dec in two successive states. By an abuse of notation, we write such a transition invariant using the $\square$ operator, and primed and non-primed versions of the process variables.

Typically the correctness argument for consensus solving algorithms is centered around the existence of *majority of processes* that support a decision. For example, the safety invariant Inv_s in Figure 1, defines a majority by the comprehension M whose cardinality is greater than $2N/3$, where p is a member of M iff $\mathsf{x}(p) = \mathsf{in}(q)$, where x is the function symbol associated with the local variable x, in the variable in of some process q, and N is the number of processes.

Furthermore, communication predicates are also defined using set comprehension and cardinality constraints over variables of type Id or $\mathsf{Set}<\mathsf{Id}>$. For example the communication predicate assumed by *OTR* in Figure 1 defines the set K of processes that all processes hear from, using the set comprehension where a process q is a member of K iff q is in the HO set of any process in the network, i.e., $\forall t.\, q \in \mathsf{HO}(t)$.

4.1 Transferring PSync specifications over asynchronous executions

The verification engine of PSync proves that the specification of a program A holds for all its runs. Nevertheless, if the specification meets several syntactic restrictions it is also true in the asynchronous executions of $\tilde{A}$. This result follows from the indistinguishability relation between the synchronous runs of A and the asynchronous ones. For presentation reasons we define only the transfer of LTL properties of the form $\square\varphi$ and $\lozenge\varphi$.

PSync has a lockstep semantics, which executes atomically the send, respectively the update, operations. Therefore, only properties of variables that are assigned at most once in (every path of) any **UPDATE** operation can be transferred on the asynchronous executions (the send operations do not modify any process variables).

Let φ be a formula from $\mathbb{CL}$ in prenex normal form, i.e., $\varphi = Q\vec{p}.\,\phi$, where Q is a sequence of quantifiers and ϕ is a conjunction of literals

Proposition 4.1. If a PSync program A satisfies a property $\square Q\vec{p}.\,\phi$ or $\lozenge Q\vec{p}.\,\phi$, then the asynchronous implementation $\tilde{A}$ satisfies the property $\square Q\vec{p}.\,\tilde{\phi}$, where $\tilde{\phi} = \bigwedge_{p,q \in \mathsf{Id}(\phi)} \mathsf{rnd}(p) = \mathsf{rnd}(q) \Rightarrow \phi$, with $\mathsf{Id}(\phi)$ denoting all the processes from ϕ.

We recall that the satisfaction relation of formulas in the assertion logic over asynchronous executions interprets process variables only over non-crashed processes (Def. 3.1).

Moreover, if the variables in the specification do not change their values during the entire computation, the specification is met also by the asynchronous implementation.

Proposition 4.2. Let $\square Q\vec{p}.\,\phi$ or $\lozenge Q\vec{p}.\,\phi$ be a property in the specification of a PSync program A, where all program variables x appearing in ϕ are assigned at most once in any **UPDATE** operation. If the program A satisfies the irrevocability property for every function symbol x in ψ, i.e., $\square\forall p.\,\mathsf{x}(p) \neq\ ? \Rightarrow \mathsf{x}(p) = \mathsf{x}'(p)$, then A satisfies $\square\varphi$, resp. $\lozenge\varphi$, implies that $\tilde{A}$ satisfies $\square\tilde{\varphi}$, resp. $\lozenge\tilde{\varphi}$, where $\tilde{\varphi} = Q\vec{p}.\,\big(\bigwedge_{\mathsf{x}\in\psi} \mathsf{x}(p) \neq ?\big) \Rightarrow \psi$.

Since the PSync program *OTR* satisfies its specification, Prop. 4.2 implies that the asynchronous implementation of *OTR* solves consensus as well.

4.2 Methodology

The verification of a PSync algorithm w.r.t. its specification from the SPEC module is based on the inductive invariant checking technique. We assume that each program is annotated with inductive invariant(s) listed in the PROP module.

Proving safety properties. Proving that a PSync program satisfies a *state* safety property $\square Safe$ uses the first invariant Inv declared after PROP. The verification engine of PSync checks that (1) Inv holds in the initial state, (2) Inv is inductive w.r.t. every lockstep, and (3) Inv implies $Safe$.

The verification conditions corresponding to the inductiveness of an invariant Inv are of the form $Inv \wedge TR \Rightarrow Inv'$, where TR is a formula relating global states before and after a lockstep, and Inv' is the invariant over primed variables states that Inv is preserve by TR.

The second invariant declared after PROP is an inductive *transition* invariant used to prove that the program satisfies the safety properties characterizing the transitions of the program, e.g., irrevocability. The verification conditions are similar to the ones generated for state safety properties.

$$
\begin{array}{rcl}
TR_{body}(\mathsf{x} := \mathsf{E})(\vec{x}(p), \vec{x'}(p)) & ::= & x'(p) = E \\
TR_{body}(\mathsf{send\ x\ to\ c})(\vec{x}(p), \vec{x'}(p)) & ::= & \mathsf{OutBox}'(p) = \mathsf{OutBox}(p) \cup \{(p, c, (x(p), \mathsf{rnd}(p))\} \\
TR_{body}(\mathsf{send\ x\ to\ all})(\vec{x}(p), \vec{x'}(p)) & ::= & \mathsf{OutBox}'(p) = \mathsf{OutBox}(p) \cup \{(p, q, (x(p), \mathsf{rnd}(p))) \mid q \in \Pi\} \\
TR_{body}(S_1; S_2)(\vec{x}(p), \vec{x'}(p)) & ::= & TR_{body}(S_1)(\vec{x}(p), \vec{x_1}(p)) \wedge TR_{body}(S_2)(\vec{x_1}(p), \vec{x'}(p)) \\
TR_{body}(\mathsf{if}\, B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2)(\vec{x}(p), \vec{x'}(p)) & ::= & (B \wedge TR_{body}(S_1)(\vec{x}(p), \vec{x'}(p))) \vee (\neg B \wedge TR_{body}(S_2)(\vec{x}(p), \vec{x'}(p))) \\
TR_{\mathsf{sync\_}\mathbf{ENV}}(\vec{x}, \vec{x'}) & ::= & \forall p.\, \mathsf{HO}'(p) = \emptyset \wedge \mathsf{OutBox}'(p) = \emptyset \\
TR_{\mathsf{sync\_}\mathbf{SEND}}(\vec{x}, \vec{x'}) & ::= & \forall p.\, \mathsf{HO}'(p) \subseteq \Pi \\
TR_{\mathsf{sync\_}\mathbf{UPDATE}}(\vec{x}, \vec{x'}) & ::= & \forall p.\, \mathsf{Mbox}'(p) = \{(v, q) \mid q \in \mathsf{HO}(p) \wedge \mathsf{rnd}(p) = \mathsf{rnd}(q) \wedge (q, p, (v, \mathsf{rnd}(q))) \in \mathsf{OutBox}(q)\} \\
 & & \wedge\ \mathsf{rnd}'(p) = \mathsf{rnd}(p) + 1
\end{array}
$$

Figure 6. Rules for computing the transition relation associated with a PSync program, where the unprimed, respectively primed variables, denote the values of the variables before respectively after the transition.

Proving liveness properties. Liveness properties $\Diamond Live$, e.g., termination in consensus, hold in any run satisfying the communication predicates. The verification conditions generated in this case depend on the communication predicates. Roughly, to ensure progress the communication predicates impose the existence of several *good* rounds where processes receive enough messages to update their local state. Our proof strategy asks the user to provide additional inductive invariants that describe the state between two good rounds.

In the following we discuss the verification strategy for communication predicates of the form $\Diamond(\varphi_1 \wedge \Diamond\varphi_2)$ In this case the user must provide two additional assertions Inv_1 and Inv_2, describing the states reached by the program after the moment when φ_1, resp., φ_2 hold. Concretely, we generate the following verification conditions:

$$
\begin{array}{lr}
Inv \wedge TR(\varphi_1) \Rightarrow Inv_1' & (Inv_1 \text{ holds after } \varphi_1) \\
Inv_1 \wedge TR \Rightarrow Inv_1' & (Inv_1 \text{ is inductive}) \\
Inv_1 \wedge TR(\varphi_2) \Rightarrow Inv_2' & (Inv_2 \text{ holds after } \varphi_2) \\
Inv_2 \wedge TR \Rightarrow Inv_2' & (Inv_2 \text{ is inductive}) \\
Inv_2 \Rightarrow Live & (Inv_2 \text{ implies } Live)
\end{array}
$$

where TR is a $\mathbb{CL}$ formula expressing the transition relation associated with a round, i.e., $TR = TR_{\mathbf{ENV}} \wedge TR_{\mathbf{SEND}} \wedge TR_{\mathbf{UPDATE}}$, and $TR(\varphi)$ is the transition relation of a round whose environment transition satisfies φ.

When communication predicates are $\Diamond(\varphi_1 \wedge \bigcirc\varphi_2)$, the user must provide only one additional inductive invariant Inv_1 describing the states reached after $\varphi_1 \wedge \bigcirc\varphi_2$ holds. The verification condition generation can be extended to deal with conjunctions of communication predicates of the two forms discussed above.

4.3 Computing verification conditions

The generation of verification conditions requires to automatically computing the transition relation associated with each round operation. The first statement of each round operation is executed synchronously by all processes and it affects several local states at once. The rest of the statements affect only the local state of the process which is executing them. Therefore, each of the formulas TR_{op} with $op \in \{\mathbf{SEND}, \mathbf{UPDATE}\}$ is defined by:

$$TR_{op}(\vec{x}, \vec{x'}) = TR_{\mathsf{sync\_}op}(\vec{x}, \vec{x_1}) \wedge \forall p.\ TR_{body}(\vec{x_1}(p), \vec{x'}(p))$$

where TR_{body} refers only to the local variables of process p (formally, it is parametrized by terms denoting these local variables, i.e., $x(p)$ with x a function symbol in $\vec{x_1}$ or $\vec{x'}$). Figure 6 gives the axioms for computing TR_{body} for the main statements executed by a program within one operation, and also the axioms defining the transition relation $TR_{\mathsf{sync\_}op}$ associated with the synchnozation marking the beginning of a new operation *op*.

Finally, we define the transition relation that takes into consideration the assumptions given by the PRED module. Let φ be a $\mathbb{CL}$ formula that describes the assumptions on the environment transition of a certain round. Then, the transition relation of that round is $TR(\varphi)(\vec{x}, \vec{x'}) = TR_{\mathbf{ENV}}(\vec{x}, \vec{x_e}) \wedge \varphi(\vec{x_e}) \wedge TR_{\mathbf{SEND}}(\vec{x_e}, \vec{x_s}) \wedge TR_{\mathbf{UPDATE}}(\vec{x_s}, \vec{x'})$.

The verification conditions are implications between $\mathbb{CL}$ formulas that combine the invariants and the transition relation as defined in Sec. 4.2. The logic $\mathbb{CL}$ is equipped with a semi-decision procedure for implication which we have used to discharge the verification conditions.

5. PSync implementation

We have implemented PSync as a domain specific language hosted in the SCALA programming language. The language is designed as a shallow embedding. The parts that require code generation, e.g. resource and state encapsulation, are handled using macros [Burmako 2013] which manipulate the SCALA syntax tree. To write an algorithm in PSync, one needs to extends the main class of the embedding called `Algorithm`. An algorithm can be passed to the runtime for execution or to the verification engine. A PSync program interfaces with an user application though the runtime.

Runtime. Figure 7 shows the different components of the system. The user is responsible for providing a PSync program and a default message handler. The state of the program is maintained within the runtime and is not externally visible. For an user, the interactions with the runtime

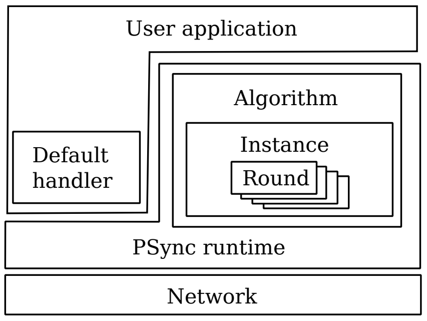

Figure 7. Runtime architecture of PSync

are asynchronous through the default handler and callback methods passed in the entry. For instance, a consensus algorithm calls a `decide` method to inform the user application of the decision. The default handler is a callback for the messages which have are not expected by any of the running instance of the algorithm. The typical reaction to such message is to start an instance of the algorithm.

The runtime is built on top of the NETTY [net] framework for event-driven network applications. The end result is asynchronous concurrent code running on top of NETTY. For transport, we choose to support UDP since a partially synchronous model is already resistant to messages loss. The serialization of data uses the pickling library [Miller et al. 2013]. The runtime can execute many instances of the same program in parallel by tagging message with an instance number. The HO sets are ghost variables used for verification purpose and are not present in the runtime.

Building a consensus service using PSync. The consensus is the classic agreement problem in distributed systems and consensus algorithms are widely used [Burrows 2006; Hunt et al. 2010; Isard 2007]. Although it is a well known problem it is still an active research area [Moraru et al. 2013]. We have build a consensus service using PSync. Here we highlight some design decisions.

Starting and stopping instances. All the replicas run a background service and consensus instances are started in a lazy way. A replica that receives a client request starts an instance of consensus. The other replicas start the instance in reaction to receiving a message via the default handler and extracts the initial value from that message.

Most of the algorithms from the literature do not actually terminate (as consensus is unsolvable in an asynchronous system). They continue running even after taking a decision. But in a real system, we cannot let instances run forever, they need to terminate and free resources. We terminate instances after they make a decision and keep a bounded log of the most recent decision. Replicas which have already terminated detect messages from the late replicas and send them the decision. Either the replica sending the recovery message still has the decision in its log, in which case it just sends it, or it sends the current state including the number of the most recent decision. In that paradigm, PSync is used to establish the global agreement, and replica are free to terminate as soon as they have witnessed agreement. However, it means that not every replica is able to witness agreement.

Recovery. To deal with crashes, we use the communication views [Chockler et al. 2001], i.e., the set of replicas connected to the system. Rather than having a built-in procedure, we use consensus to agree on view changes. Our system supports adding and removing replicas on the fly. However, to guarantee correctness, the view is constant within an instance of the algorithm. When an instance starts it takes a snapshot of the current view. Any modification of the view is visible only for the instance starting after the modification.

Verification. For the verification, we have implemented a semi-decision procedure based on [Dragoi et al. 2014]. During the compilation, we automatically extract $\mathbb{CL}$ formulas corresponding to the transition relation of the PSync program. The verification conditions are generated from this information and discharged at a later time. We uses Z3 [Moura and Bjorner 2008] as backend prover.

6. Evaluation

We evaluate our implementation of PSync with two different consensus algorithms with different communication pattern to see how they fare in a partially synchronous abstraction. Then, we evaluate the system under different types of faults, machine and network. Finally, we report about the verification of the algorithms.

We use a consensus algorithm to order write requests to a simple key-value store. Two requests operating on separate keys are allowed to proceed in parallel. On the other hand, for the same key, a new request is allowed to proceed only after the previous request has terminated. We use 50 keys and queries are generated randomly. This gives an expected conflict rate of 2%. This setting is inspired from [Moraru et al. 2013, Section 7.1].

The algorithms. We chose two consensus algorithms to evaluate our system. Our choice was guided by the fact that these two algorithms are based on different principles (size of the quorum, communication pattern). Both algorithms comes from [Charron-Bost and Schiper 2009].

One Third Rule (OTR) is described in more details in Example 2.1. At every round, each replica sends a message to all the replicas. The algorithm uses a two thirds majority. Therefore, four replicas are required to resist one crash. The implementation of the algorithm (excluding the specification) is about 30 lines of code.

Last Voting (LV) is an encoding of Paxos [Lamport 1998] in the HO-model. The algorithm uses a rotating coordinator. Depending on the round either the coordinator sends message to every process or every process send to the coordinator. The algorithm establish a strict majority of process agreeing before taking a decision. Therefore, three replicas are required to resist one crash. The implementation of the algorithm (excluding the specification) is about 90 lines of

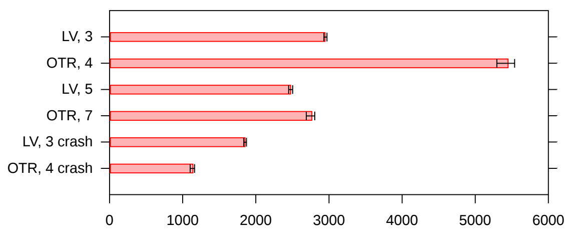

Figure 8. Throughput (instances / sec.). The labels indicate the algorithm, number of replicas, and if a replica is crashed.

code. We allow the process to specify how many messages are expected during a round, this allows the coordinator to make progress as soon as it receives $n/2 + 1$ messages.

Performance evaluation. We run our first set of experiments on servers with Intel Xeon X5460 cpu, 8 GB ram, and running openJDK 1.6. We report throughput numbers on configurations tolerating 1,2 crashes. For the OTR this corresponds to 4,7 replicas and 3,5 replicas for the LV. The typical network RTT is less than 1 ms.

The throughput is computed from the number of instances of the algorithms executed during a 2 minutes period. Each experiment is run 10 times. We report the average, minimum, and maximum throughput. The results are shown in Figure 8. We can observe that the OTR give better performance in a configuration tolerating 1 crash. On the other hand, LV since is uses a number of messages linear in the number of replicas (square for OTR) scales better.

Resilience to crashes. In this experiment, we evaluate the behavior of the two algorithms when a machine crashes. We run 3 replicas instead of 4 for the OTR and 2 instead of 3 for the LV. The results are shown in Figure 8. The OTR suffers from a significant performance hit as each round progress only when it reaches the timeout. On the other hand, the LV is not affected as much by the crash. This is explained by the fact that the LV can make some progress as soon as it receives more than half of the messages.

Resilience to network faults. We also evaluate our system against a network dropping packets. To have a precise control over the rate at which the network is dropping packets, we run this experiment with all the replicas on the same machine and use Linux network emulator[4] to drop packets. We evaluate the system with 0,1,2,5,10% of the packet dropped. In this experiment, all the replicas run on the same machine which has a Intel Core i7-4700 cpu, 8 GB of ram, runs the openJDK 1.7. The results are shown in Figure 9. We observe that in the PSync setting packet loss affects the algorithms in a similar way even though both algorithms have different communication pattern.

Verification. We verified the OTR and LV algorithms. Both the specification, consensus, and the invariants are provided by the user. In table 1, we report the number of invariants

[4] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`

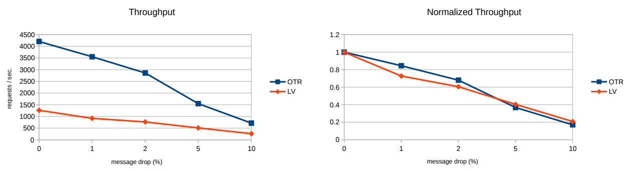

Figure 9. Throughput against packet loss in a configuration tolerating one crash.

Algorithm (LOC)	#Invariants (LOC)	#Vcs	Solving time in s.
OTR (30)	4 (23)	27	5
LV (85)	8 (35)	45	16

Table 1. Verification results

provided by the user, the number of verification condition and the time to solve them. The number of invariants includes the includes the specification of auxilliary functions. We also inlcude the number of lines of code needed for the algorithms and the specification.

7. Related work

There exists multiple formulations of partially synchronous models [Dwork et al. 1988; Gafni 1998; Biely et al. 2007; Charron-Bost and Schiper 2009; Afek and Gafni 2013].

Implementing fault tolerant systems is difficult [Chandra et al. 2007] and many different algorithms [Lamport 1983; Ben-Or 1983; Lamport 1998; Brasileiro et al. 2001; Mao et al. 2008; Moraru et al. 2013; Widder et al. 2012] exists. There even are algorithms, such as Raft [Ongaro and Ousterhout 2014] for the consensus problem, that have been developed with the purpose of being understandable. We think that the problem does not comes form the algorithm but from the way of thinking about the system. Therefore, we adopt an higher level abstraction to implement fault-tolerant algorithms.

On the verification side, there are automated techniques that work on a finite number of processes [Tsuchiya and Schiper 2007, 2008, 2011; Lamport 2000] or mechanized techniques based on interactive proofs assistants [Charron-Bost and Merz 2009; Debrat and Merz 2012; Chaudhuri et al. 2008]. However, the focus of those work is on proofs at the level of models not implementation.

Exploiting synchronous behaviors of asynchronous systems for verification has been explored in [Basu et al. 2012] and [Desai et al. 2014]. [Basu et al. 2012] propose the notion of synchonizability, a synchonizable system has the same synchronous and asynchronous behaviour. [Desai et al. 2014] propose look at invariants where the communication channels are almost empty. Our approach is different as we start with a synchronous abstraction rather than retrofiting synchrony in existing asynchronous systems. Also in terms of we consider the parametric verification problem instead of systems with a finite number of processes.

References

The netty project. http://netty.io/. Accessed: 2014-07-03.

Y. Afek and E. Gafni. Asynchrony from synchrony. In D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, and P. Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, pages 225–239, 2013. doi: 10.1007/978-3-642-35668-1_16. URL http://dx.doi.org/10.1007/978-3-642-35668-1_16.

K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986. doi: 10.1016/0020-0190(86)90071-2. URL http://dx.doi.org/10.1016/0020-0190(86)90071-2.

S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 56–71, 2012. doi: 10.1007/978-3-642-27940-9_5. URL http://dx.doi.org/10.1007/978-3-642-27940-9_5.

M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC*, pages 27–30. ACM, 1983. doi: 10.1145/800221.806707.

M. Biely, B. Charron-Bost, A. Gaillard, M. Hutle, A. Schiper, and J. Widder. Tolerating corrupted communication. In *PODC*, pages 244–253, 2007.

F. Borran, M. Hutle, and A. Schiper. Timing analysis of leader-based and decentralized Byzantine consensus algorithms. *J. Braz. Comp. Soc.*, 18(1):29–42, 2012.

F. Brasileiro, F. Greve, A. Mostefaoui, and M. Raynal. Consensus in one communication step. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 2127 of *Lecture Notes in Computer Science*. Springer, 2001. ISBN 978-3-540-42522-9. doi: 10.1007/3-540-44743-1_4.

E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. doi: 10.1145/2489837.2489840. URL http://doi.acm.org/10.1145/2489837.2489840.

M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: 10.1145/1281100.1281103. URL http://doi.acm.org/10.1145/1281100.1281103.

B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA+ proof system. In *LPAR Workshops*, 2008.

G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, Dec. 2001. ISSN 0360-0300. doi: 10.1145/503112.503113. URL http://doi.acm.org/10.1145/503112.503113.

H. Debrat and S. Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012, 2012.

A. Desai, P. Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 709–725, 2014. doi: 10.1145/2660193.2660211. URL http://doi.acm.org/10.1145/2660193.2660211.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975. URL http://doi.acm.org/10.1145/360933.360975.

D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34:77–97, 1987. http://doi.acm.org/10.1145/7531.7533.

C. Dragoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In K. L. McMillan and X. Rival, editors, *VMCAI*, pages 161–181. Springer, 2014.

C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, Apr. 1988.

T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3): 155–173, 1982.

M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2): 374–382, Apr. 1985.

E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In B. A. Coan and Y. Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 143–152, 1998. doi: 10.1145/277697.277724. URL http://doi.acm.org/10.1145/277697.277724.

P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC*. USENIX Association, 2010.

M. Isard. Autopilot: Automatic data center management. *Operating Systems Review*, 41(2):60–67, April 2007. URL http://research.microsoft.com/apps/pubs/default.aspx?id=64604.

I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *PODC*, pages 169–178, 2006.

L. Lamport. The weak Byzantine generals problem. *J. ACM*, 30(3): 668–676, July 1983. ISSN 0004-5411. doi: 10.1145/2402.322398.

L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998. ISSN 0734-2071.

L. Lamport. Distributed algorithms in TLA (abstract). In *PODC*, 2000.

L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855767.

H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *OOPSLA*, pages 183–202, 2013.

I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2517350. URL http://doi.acm.org/10.1145/2517349.2517350.

L. Moura and N. Bjorner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.

D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

T. Tsuchiya and A. Schiper. Model checking of consensus algorithms. In *SRDS*, pages 137–148, 2007.

T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In *DISC*, pages 466–480, 2008.

T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6): 341–358, 2011.

J. Widder, M. Biely, G. Gridling, B. Weiss, and J.-P. Blanquart. Consensus in the presence of mortal Byzantine faulty processes. *Distributed Computing*, 2012.