PSYNC: A partially synchronous language for
fault-tolerant distributed algorithms
Cezara Drăgoi, Thomas Henzinger, Damien Zufferey

To cite this version:
Cezara Drăgoi, Thomas Henzinger, Damien Zufferey. PSYNC: A partially synchronous language for
fault-tolerant distributed algorithms. POPL ’16 - 43rd Annual ACM SIGPLAN-SIGACT Symposium
on Principles of Programming Languages, Jan 2016, Saint Petersburg, Florida, United States. pp.400-
415, 10.1145/2837614.2837650. hal-01251199

HAL Id: hal-01251199
https://inria.hal.science/hal-01251199
Submitted on 5 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
PSYNC: A partially synchronous language for fault-tolerant distributed algorithms

Cezara Drăgoi  
INRIA, ENS, CNRS  
cezara.dragoi@inria.fr

Thomas A. Henzinger *  
IST Austria  
tah@ist.ac.at

Damien Zufferey †  
MIT CSAIL  
zufferey@csail.mit.edu

Abstract

Fault-tolerant distributed algorithms play an important role in many critical/high-availability applications. These algorithms are notoriously difficult to implement correctly, due to asynchronous communication and the occurrence of faults, such as the network dropping messages or computers crashing.

We introduce PSYNC, a domain specific language based on the Heard-Of model, which views asynchronous faulty systems as synchronous ones with an adversarial environment that simulates asynchrony and faults by dropping messages. We define a runtime system for PSYNC that efficiently executes on asynchronous networks. We formalise the relation between the runtime system and PSYNC in terms of observational refinement. This high-level synchronous abstraction introduced by PSYNC simplifies the design and implementation of fault-tolerant distributed algorithms and enables automated formal verification.

We have implemented an embedding of PSYNC in the SCALA programming language with a runtime system for partially synchronous networks. We show the applicability of PSYNC by implementing several important fault-tolerant distributed algorithms and we compare the implementation of consensus algorithms in PSYNC against implementations in other languages in terms of code size, runtime efficiency, and verification.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Verification

Keywords Fault-tolerant distributed algorithms, Round model, Partially synchrony, Automated verification, Consensus

1. Introduction

The need for highly available data storage systems and for higher processing power has led to the development of distributed systems.

* Check conclusions This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RISE) and Z211-N23 (Wittgenstein Award).
† ToDo ...

A distributed system is a set of independent nodes in a network, that communicate and synchronize via message passing, giving the illusion of acting as a single system. The difficulty in designing these systems comes from the network unreliability and the host failures: messages can be dropped and nodes can crash. The processes have only a limited view over the entire system and they must coordinate to achieve global goals.

A general mechanism for implementing a fault-tolerant service is replication: the application is copied on different replicas which are kept consistent. Clients send requests to a replica running the service, the service communicates with the other nodes to maintain a consistent state of the global system, and replies to the clients. Consistency is maintained by solving consensus problems. Each process has an initial value, and all processes have to agree on a unique decision value among the initial ones. Therefore all replicas return the same value when queried for the attribute of an object. Consensus algorithms have received a lot of attention in academia and industry, because they are at the core of most high-availability systems, but are difficult to design and implement. Because consensus is not solvable in asynchronous networks in the presence of faults [35], a large number of algorithms have been developed [20, 31, 44, 47, 53, 59], each of them solving consensus under different assumptions on the type of faults, and the degree of synchrony of the system. Moreover, many other problems in distributed systems can be reduced to agreement, for example, atomic broadcast has been shown equivalent with consensus. Note-worthy examples of applications from industry that use consensus algorithms include the Chubby lock service [18], which uses the Paxos [47] consensus algorithm, Apache Zookeeper which has a dedicated algorithm [44] for primary-backup replication.

In this paper we unify the modeling, programming, and verification of fault-tolerant distributed algorithms with a domain specific language, called PSYNC, that

- has a high-level, round-based, control structure that helps the programmer focus on the algorithmic questions rather than spending time fiddling with low-level network and timer code;
- compiles into working, efficient systems that preserve the important user-defined properties of high-level algorithms;
- is amenable to automated verification.

Despite the importance of fault-tolerant algorithms, no widely accepted programming model has emerged for these systems. The algorithms are often published with English descriptions, or, in the best case, pseudo-code. Moreover, fault-tolerant algorithms are rarely implemented as published, but modified to fit the constraints and requirements of the system in which they are incorporated [21]. General purpose programming languages lack the primitives to implement these algorithms in a simple way. A developer is forced to choose between low-level libraries, e.g., POSIX, to write network
and timer code, or high-level abstractions, like supervision trees, that come with their own limitations. The first approach leads to programs with a very complex control structure that hides the algorithm’s principles. The later may not work because the abstraction’s assumptions do not fit the system’s requirements.

The complexity of fault-tolerant implementations makes them prone to error so they are prime candidates for automated verification. Their complexity is twofold: (1) the algorithms these implementations are based on, have an intricate data-flow and (2) the implementations have complex concurrent control structure, e.g., an unbounded number of replicas communicate via event-handlers, and complex data structures, e.g., unbounded buffers. Although fault-tolerant algorithms are at the core of critical applications, there are no automated verification techniques that can deal with the complexity of an implementation of an algorithm like Paxos.

The standard programming paradigm for implementing fault-tolerant distributed algorithms requires reasoning about asynchrony and faults separately. Programming languages provide access only to asynchronous communication primitives. A fundamental result on distributed algorithms [35] shows that it is impossible to reach consensus in asynchronous systems where at least one process might crash. Therefore the algorithms that solve consensus make assumptions on the network finer than asynchrony, i.e., they need to reason about time explicitly. In order to reconcile the modeling of various assumptions on network, the algorithms community introduced computational models that uniformly model (a)synchrony and faults using an adversarial environment that drops messages [24, 37]. We take a programming language perspective on this matter and propose a domain specific language, PSYNC, that offers the programmer the illusion of synchrony, uses the adversarial environment to reason about faults and asynchrony, and efficiently executes on asynchronous networks.

High-level computational model. PSYNC is based on the Heard-Of model [24], which structures algorithms in communication-closed rounds [32]. A PSYNC program is defined by a sequence of rounds, and has a lockstep semantics where all the processes execute the same round. Each round consists of two consecutive operations: (1) a send method for sending messages and (2) an update method that updates the local state according to the messages received during the round. A round is communication-closed if all the messages are either delivered in the round they are sent or dropped. Communication-closed rounds provide a clear scope for reasoning about time invariants. Compared with asynchronous programming models, the lock-step semantics leads to a smaller number of interleavings and simpler inductive characterizations of the set of reachable states. Lock-step semantics leads to a smaller number of interleavings and simpler inductive characterizations of the set of reachable states.

In the HO-model, a distributed system is a set of processes together with an adversarial environment, where the environment determines the set of messages received in a round. Each process has a Heard-Of set, denoted HO, which is a variable over sets of process identities exclusively under the control of the environment. HO-sets abstract the asynchronous and faulty behaviors of the network. In a given round, process p receives a message from process q if q sends a message to p and q ∈ HO(p). The network’s degree of synchrony and the type of faults correspond to assumptions on how the environment picks the HO-sets.

The round structure together with the HO-sets define a notion of abstract time. Using communication-closed rounds, PSYNC introduces a high-level control structure that allows the programmer to focus on the data computation performed by each process to ensure progress towards solving the considered problem, as opposed to spending time on the programming language constructs, e.g., incrementing message counters, setting timers.

Runtime. We have implemented PSYNC as an embedding in Scala. PSYNC programs link against a runtime system that manages the interface with the network and the resources used by the PSYNC program. The code runs on top of an event-driven frame-

work for asynchronous network applications, uses UDP to transmit data, in a context where processes may permanently crash. The main challenge in deriving asynchronous code from a PSYNC program is defining a procedure that decides when to go to the next round while allowing sufficiently many messages to be delivered. Our approached is based on timeouts: roughly during an execution the time spend by a process to receive messages is divided into chunks of equal length, i.e., the timeout. Each chunk corresponds to one round, identifying uniquely set of messages received by the process within that round. The timeout is a parameter that influences performance and can be fine-tuned to take advantage of the network and the algorithm specificity.

We show that for any PSYNC program P the asynchronous executions of P generated by the runtime are indistinguishable from the lock-step executions of P. Indistinguishability is a relation between executions which roughly says that no process can locally distinguish between them. Intuitively, this relation is important because it is not possible to solve a problem that requires different answers for executions which are indistinguishable from the local perspective of processes.

Using indistinguishability we show that the asynchronous system defined by the runtime of a program P observationally refines the synchronous system defined by P, that is every behavior of a client that uses the runtime can be reproduced if the client uses P instead, provided that the client operations are commutative.

Verification. Due to the complexity distributed systems have reached, we believe it is no longer realistic nor efficient to assume that high level specifications can be proved when development and verification are two disconnected steps in the software production process. PSYNC provides a simple round structure whose the lockstep semantics leads to a smaller number of interleavings and simpler inductive characterizations of the set of reachable states. We have identified a large class of specifications which if met by the lock-step semantics of PSYNC then they are also met by its asynchronous semantics used at runtime.

We have implemented a state-based verification engine for PSYNC, that checks safety and liveness properties. The engine assumes the program annotated with inductive invariants and ranking functions, and proves the validity of these annotations and the fact that they imply the specification. In general annotating a program with inductive invariants is a hard task even for an expert. The advantage of PSYNC lies in the simplicity of the required inductive invariants. Compared with an asynchronous programming model, the round structure allows looking at the system’s invariants at the boundary between rounds and ignores messages.

Contributions. We introduce PSYNC a domain specific language for fault-tolerant algorithms.

- PSYNC makes it possible to write, execute, and verify high-level implementations of fault-tolerant algorithms.
- PSYNC has a simple synchronous semantics but PSYNC programs run on asynchronous networks using a runtime system whose executions are indistinguishable from those of PSYNC.
- We prove that for any program P, the runtime of P observationally refines P, assuming clients with commutative operations.
- We prove that, for an important class of specifications including consensus, if a PSYNC program satisfies the specification, then its runtime system satisfies it as well.
- We have implemented and verified several fault-tolerant algorithms using PSYNC. We evaluate the LastVoting [24] that corresponds to Paxos [47] in the HO-model, in a distributed key-value store, and show that the PSYNC implementation performs comparably to high-performance consensus implementations.
A simplified version of LastVoting in PSYNC. The program has four rounds, which execute in a loop. \( r \) contains the round number. The function \( \text{coord}(r) \) returns the identity of the coordinator of round \( r \). Its identity changes between phases. In one phase, the coordinator collects proposals from the other replicas, picks one of them (Collect), and tries to impose it to the other replicas (Candidate). If a majority of processes agree with the coordinator’s proposal (Quorum) then eventually all processes will accept this value as their decision (Accept).

Sec. 2 introduces PSYNC using an example. Sec. 3 defines indistinguishability and observational refinement. Sec. 4 defines the domain specific language PSYNC, while Sec. 5 describes its runtime system. The verification techniques are presented in Sec. 6. Finally, Sec. 7 presents the experimental evaluation of PSYNC.

2. Overview

In this section we present the main features of PSYNC using a consensus service. The service uses the LastVoting [24] algorithm, shown in Fig. 1, an adaptation of the Paxos algorithm to the HO-model.

We want to build a distributed read-write register. The application needs to give the users a coherent view of the data. Read requests can be served locally, but write requests need to be agreed upon globally. To agree on the order of requests, or sets of requests (batching), the application uses a consensus service.

The consensus service exports event-driven operations that the clients use to communicate with the service. These operations define the interface of the service. The interface of a service implemented in PSYNC has two main types of operations: input operations, denoted \( \text{init} \), that the clients use to send a request to the service, and output operations, denoted \( \text{out} \), that the service uses to reply to a client request.

For example, a client sends a request to the LastVoting program using \( \text{init}(v) \), where \( v \) is the new value the client wants to write to the register. The service replies to the client by generating an output event \( \text{out}(v') \) which contains the new value of the register.

A service process that receive a client request, starts executing LastVoting by calling its initialization function \( \text{init} \). We assume that the clients are constantly sending new requests so each process has an initial value. Each replica stores the value it believes should be written in the variable \( x \). After the initialization phase replicas typically have different \( x \) values. The goal of the algorithm is to make the replicas agree on one of these values. The execution of LastVoting terminates when all replicas agree.

LastVoting roughly works by first establishing a majority of processes that agree on the same value. A designated process, called the coordinator, collects proposals for the value of \( x \) from the other replicas and picks one of them (execution of the round Collect, Line 2). In the next round, Candidate (Line 9), the coordinator tries to impose the chosen value to a majority of replicas. The round Quorum (Line 17) checks that this majority has been correctly established. If a quorum is formed then a decision is made, and all replicas will accept it as the decision value, during the Accept round (Line 24). The execution of these rounds repeats, starting again with the Collect round, possibly with a different coordinator. The crux for the correctness of the algorithm is that strict majorities have a non-empty intersection. Since a decision requires the agreement of a majority of processes, the decision is unique.

Program structure. A PSYNC program is composed of an interface, a set of local variables, an initialization function, and a sequence of rounds. Each round defines a send and an update method. All processes execute in lockstep the same round. A part from the local variables processes use build-in read-only variables: \( r \) represents the round number, \( n \) the number of processes in the system, and \( id \) is the process unique identifier.

The \text{init} method takes as argument an integer. The \text{send} function returns the set of messages sent by the process executing it during the current round. A message is a pair (payload, recipient). The payload type might differ across rounds. For example, in Collect processes send pairs of integers while in Quorum they send one integer. The recipient is identified by a ProcessId. PSYNC supports broadcast and point-to-point communication. The statement \text{broadcast}(T) returns a set of messages of type \( T \), ProcessId, where \( T \) is the type of payload. This set contains one message for every process in the system (all messages have the same payload).
The \textbf{update} takes as argument the set of messages received during the current round and modifies the local state of the process. The \textbf{input} of \textbf{update} is a set of pairs of type $\langle T, \text{ProcessId} \rangle$, where $T$ is the type of the payload and the second component is the sender’s identity. The \textbf{update} method can use external functions to perform sequential computations and modify the local state. For example, $\text{mbox.valWithMaxTs}$ used in \textbf{Collect} scans the set of received messages and returns a value $v$ such that $(v, t)$ belongs to $\text{mbox}$ and for any other $(v', t')$ in $\text{mbox}$, $t' \leq t$. The functions $\text{collect}$ and $\text{Accept}$ project a pair on its first resp. second component.

\textbf{Execution.} A run of a PSYNC program starts with a call to the initialization function \textbf{init} (on each process). In \textbf{LastVoting} the \textbf{init} function initializes the value of $x$ known locally by each replica. The run continues with processes repeatedly executing, in lock step, the sequence of rounds defined by phase in the program.

In an ideal system (where no messages are lost or delayed) an execution of one phase of the \textbf{LastVoting} program would result in the trace shown in Fig. 2. The processes proceed in lockstep, messages are delivered in time, and agreement is reached after four rounds. In reality an execution is more likely to look like the one shown in Fig. 3a, due to different delivery times for messages, different processors speeds, and crashes.

To reason about asynchrony and faults, the PSYNC semantics is based on the HO-model. Each process has a variable interpreted over sets of processes, called HO-set. The set of messages received by a process $p$ in round $r$, is the set of messages that were sent to $p$ by the processes in its HO-set. At the beginning of each round, HO-sets are non-deterministically modified by the environment.

The HO-model captures asynchronous behaviors and faults while providing the illusion of a lockstep semantics. Therefore, it is possible to reflect the faults in Fig. 3a using a lockstep semantics by setting the HO sets to the appropriate values. Fig. 3b shows a lockstep execution where each process receives the same set of messages as in Fig. 3a. If a message is dropped by the network, e.g., the message $p_2$ sends to $p_1$ in \textbf{Collect}, then $p_2$ is not included in the HO of $p_1$. If a message is delayed too long after $p_3$ starts executing round \textbf{Accept} despite not receiving the acknowledgement sent by $p_3$. However, this acknowledgement is eventually delivered when the coordinator is in the fourth round, but the message comes too late to influence the local computation. Therefore, it is as if the message was lost. In the lockstep execution, Fig. 3b, $p_3$ is not included in the HO-set of $p_1$ in the \textbf{Quorum} round. Finally, crashes do not directly impact the view of correct processes. Crashed processes are modeled using correct process which are not included in the HO-set of any other process. It is as if all messages they send are dropped after the instant of the actual crash.

\textbf{Specification and Verification.} We have developed a state-based verification engine for PSYNC programs. The specification of \textbf{LastVoting} includes properties like agreement, which say that all processes decided on the same value:

\[
\forall p, p'. p.\text{decided} \land p'.\text{decided} \Rightarrow p.\text{decision} = p'.\text{decision},
\]

where $p, p'$ are processes and $p.\text{decided}, p'.\text{decided}$ is the value of the local variable $\text{decided}$ resp. decision, of process $p$.

The verification engine is based on deductive verification. This assumes that the program is annotated with inductive invariants and the engine proves the validity of the annotations and the fact that they imply the specification. The lockstep semantics is essential in order to have simple inductive invariants. For instance, the crux of the invariant that shows agreement is the existence of a majority of processes that agree on a value $v$ when at least one process decides:

\[
\exists v, t, A. A = \{ p | p.\text{ts} \geq t \} \land |A| > n/2 \land \forall p, x \in A \Rightarrow p.\text{x} = v.
\]

Beyond safety properties, we are also interested in proving liveness properties, such as, $\forall p. p.\text{decided}$. Because consensus is not solvable in asynchronous networks with faults, the network must satisfy liveness assumptions to ensure progress of the algorithm. The liveness assumptions impose constraints, typically lower bounds on the cardinality of the HO-set. For example, proving that \textbf{LastVoting} eventually makes a decision requires a sequence of four rounds starting with \textbf{Collect} during which the environment picks values for the HO-set such that:

\[
|\text{HO(coord(r))}| > n/2 \land \forall q. \text{coord}(r) \in \text{HO}(q).
\]

For fault-tolerant distributed algorithms, the specification, the liveness assumptions, and the inductive invariants, require reasoning about set comprehensions, cardinality constraints, and process quantification. To express these properties and check their validity we use a fragment of first-order logic, called CL, and its semi-decision procedure [30]. The complete \textbf{LastVoting} program, its specification and annotations are available in Appendix ??.

### 3. Indistinguishability and Observational Refinement

In this section we formally define in terms of transition systems the main theoretical concepts used in the paper: indistinguishability, observational refinement, and the relationship between them.

#### 3.1 Definitions

A \textit{transition system} is a tuple $TS = (P, V, A, s_0, T)$, where $P$ is a set of processes, $V$ is a finite set of variables, $V = \bigcup_{p \in P} V_p$ with $V_p$ the set of local variables of process $p \in P$, $A$ is a (possibly infinite) set of labels, $A = \bigcup_{p \in P} A_p$ with $A_p$ the set of transition labels of process $p \in P$, $s_0 \in \Sigma$ is the initial state of the system, $T$ is a transition relation $T \subseteq \Sigma \times 2^{\Sigma} \times \Sigma$ where the label of each
transition contains at most one label per process. The state space of \( TS \) is \( \Sigma = \{ P \rightarrow V \rightarrow D \} \cup \{ * \} \) where \(* \) is a special state disjoint from the others and \( D \) the data domain where variables are evaluated. A state \( s \in \Sigma \) is a valuation of the processes variables.

Given a process \( p \in P \), \( s(p) \) is the local state of \( p \), which is an evaluation of \( p \)'s local variables, i.e., \( s(p) \in [V_p \rightarrow D] \).

An execution of a system \( TS \) is an infinite sequence \( s_0,A_0,s_1,A_1, \ldots \) such that for all \( i \geq 0 \), \( s_i \in \Sigma \), \( A_i \subseteq A \) \( (A_i \neq \emptyset) \) such that each process takes at most one transition, and the state \( s_{i+1} \) is a successor of the state \( s_i \) in the transition relation \( T \), i.e., \( (s_i,A_i,s_{i+1}) \in T \). We denoted by \( [TS] \) the set of executions of the systems \( TS \). A run is the sequence of states induced by an execution and a trace the sequence of labels induced by an execution. We denoted by \( \text{Runs}(TS) \), \( \text{Traces}(TS) \), the set of runs, respectively the set of traces, of a transition system \( TS \).

The projection of execution \( \pi \) on a process \( p \), denoted \( \pi \mid_p \), is obtained from \( \pi \) by keeping only the state of the variables and transition labels local to \( p \), i.e., \( \pi \mid_p = s_0,A_1,s_1, \ldots \) where \( s_i = s_i(p) \) and \( A_i = A_i \cap A_p \).

Let \( s \in \Sigma \) be an alternating sequence over the alphabet \( \Sigma \) and \( A \). A block over \( s \) is a word \( s_i(\emptyset s)^j \) where \( s \in \Sigma \) and \( \emptyset \) is a transition with an empty label. The stuttering closure of \( \pi \) is the set of executions in the language generated by replacing every state in \( \pi \) with the corresponding block. We write \( \pi_1 \equiv \pi_2 \) iff the execution \( \pi_1 \) is equivalent to the execution \( \pi_2 \) up to stuttering, i.e., there is an execution in the stuttering closure of both \( \pi_1 \) and \( \pi_2 \).

### 3.2 Indistinguishability

We define indistinguishability, an equivalence relation between executions of a transition system. Furthermore, we lift this definition to relate executions of different transition systems.

**Definition 1** (Indistinguishability). Given two executions \( \pi \) and \( \pi' \) of a system \( TS \), a process \( p \) cannot distinguish locally between \( \pi \) and \( \pi' \), denoted \( \pi \nearrow_p \pi' \) iff the projection of both executions on the local states and transitions of \( p \) agree up to finite stuttering, i.e., \( \pi \mid_p \equiv \pi \mid_p \).

Two executions \( \pi \) and \( \pi' \) are indistinguishable, denoted \( \pi \equiv \pi' \), iff no process can distinguish between them, i.e., \( \forall p \in P \times \pi \equiv \pi \).

To relate executions of different systems indistinguishability is extended, as expected, to \( \equiv \) where \( W \) and \( L \) are subsets of the common variables and labels of the two systems.

**Definition 2** (Indistinguishable systems). A system \( TS_1 \) is indistinguishable from a system \( TS_2 \) denoted \( TS_1 \equiv TS_2 \) iff they are defined over the same set of processes and for any execution \( \pi \in [TS_1] \) there exists an execution \( \pi' \in [TS_2] \) such that \( \pi \equiv W,L \) \( \pi' \) where \( W = V_1 \cap V_2 \) and \( L = A_1 \cap A_2 \).

Notice that indistinguishability takes into account only the local process order of transitions. For example, the two executions in Fig. 3 are indistinguishable, although the Quorum round is executed during non-overlapping periods of time by process \( p_1 \) and \( p_2 \) in Fig. 3a while in Fig. 3b the order between them is lost.

### 3.3 Observational refinement

A client and a service synchronize through an interface given by the shared transition labels. When a client uses a PSYNC program as a service, e.g., the \( \text{LastVoting} \) as a consensus service, the client should be able to use indistinguishable systems interchangeably.

A distributed client of a service is a transition system defined by the parallel compositions of a set of \( n \) transitions systems over disjoint alphabets, each of this transition systems representing one client process in the system. This definition implies that transitions of different clients processes commute, which intuitively means client processes do not synchronize with each other directly. The full definition is in Appendix B.

**Composition between a client and a service.** We assume the network is formed of different nodes, such that each node consists of a client process and a service process. A client sends requests to the service process located on the same node. For presentation reasons we assume that the transitions system associated with the client respectively with the service use the same set of processes, corresponding to the nodes in the network.

Let \( S = (P,V,A \cup \{ s, T \}) \) be a service interface \( I \). A client \( C = (P,V,A,s, T) \) connects to the service \( S \) if \( I \subseteq A \) and \( A \cap A' = \emptyset \). We consider \( I = \Psi \subseteq P \), where \( I_p \) is the labels of transitions only taken by the process \( p \), in both \( S \) and \( C \).

We define the behaviors of client \( C \) that interacts with service \( S \) as the set of executions of a transition system denoted \( C(S) \). The system \( C(S) \) is obtained by (1) taking the asynchronous product between \( C \) and \( S \) and forcing that any transitions labeled by \( b \in I \) is always taken simultaneously by \( C \) and \( S \), and (2) projecting out \( S \) from the product (we are only interested in the client), (3) After projecting out \( S \), we remove the transitions with no labels. We obtain a transition system with the same state space as the original client, but with fewer behaviors.

For example, a client communicates with \( \text{LastVoting} \) in Fig. 1 using the interface \( \text{LastVoting} = (\{ \text{init}_p(v) \}, \text{out}_p(v) | p \in P, v \in Z) \). For the client, an \( \text{init}_p(v) \) transition sends a request to process \( p \) executing \( \text{LastVoting} \) and makes the client wait for the reply \( \text{out}_p(v) \). An \( \text{init}_p(v) \) transitions of \( \text{LastVoting} \) corresponds a call to the initialization function \( \text{init} \) on process \( p \) with \( v \) as input parameter. Similarly \( \text{out}_p(v) \) transitions are used to handle the replies between the service process \( p \) and the client process \( p \).

In general there are different possible implementations of a service. In the following we define a refinement relation between service implementations.

**Definition 3** (Observational Refinement). Let \( TS_1 \) and \( TS_2 \) be two transition systems and a common interface \( I \). Then, \( TS_1 \) refines \( TS_2 \) w.r.t. \( I \) denoted \( TS_1 \sqsubseteq_I TS_2 \), if for any client \( C \), \( \text{Runs}(C(TS_1)) \subseteq \text{Runs}(C(TS_2)) \).

We say that \( TS_1 \) observationally refines \( TS_2 \) if every run of a client that uses \( TS_1 \) is also a run the same client using \( TS_2 \).

Moreover, since we consider clients which do not impose an order between the transitions on different client processes, indistinguishability is equivalent with observational refinement.

**Theorem 1.** Let \( TS_1 \) and \( TS_2 \) be two systems with a common interface \( I \). If \( TS_1 \sqsupseteq TS_2 \) then \( TS_1 \sqsubseteq_I TS_2 \).

Proof. (Sketch) The proof is based on the Corollary 43 from [34], which states that sequentially consistency is equivalent with observational refinement when client transitions commute across processes. The main ingredient to extend this corollary is that indistinguishability is equivalent with sequential consistency when one of the systems is sequential. More precisely, if \( TS_2 \) is a sequential transition system, then \( TS_1 \sqsupseteq TS_2 \) iff \( TS_1 \) is sequentially consistent with \( TS_2 \).

### 4. Syntax and semantics of PSYNC

PSYNC is designed as a domain specific language embedded within a general purpose programming language. PSYNC offers specialized syntactic constructs for distributed algorithms. In this section we define only the core communication and synchronization control structures using an abstract syntax and we focus on the types of the operations composing a PSYNC program.

**Syntax.** We give an abstract syntax for PSYNC programs in Fig. 4. A program has an interface, a number of local variables, an initial-
ization operation \texttt{init}, and a non-empty sequence of rounds, called phase. Each process executes in a loop the sequence of rounds defined in the phase.

The interface is a set of functions, which are the transitions observed by the client. The interface includes an \texttt{init} event that corresponds to calling the \texttt{init} operation. A client request is received using an \texttt{init} labeled transition. The other functions are outputs of the program to the client. We denote by \texttt{name}_p, a label in the interface when the associated transition is executed by process \texttt{p}.

Each round is parameterized by the type \texttt{T} which represents the payload of the messages. The \texttt{send} and update operations of a round must agree on \texttt{T}. However, different rounds can have different payload types. For instance, the payload of \texttt{Collect} in Fig 1 is \((\text{Int},\text{Int})\) and \texttt{Int} for the other type of rounds.

The operations in a PSYNC program do not use directly any iterative control structures. For complex sequential computations they can use auxiliary operations implemented in the host language. The \texttt{init}, \texttt{send}, and \texttt{update} operations are assumed to terminate within a number of steps that depends on the number of processes and the input values of the initialization function. PSYNC is designed to facilitate the implementation of message passing concurrency. Proving total correctness of the sequential code executed by each process is orthogonal to the scope of PSYNC.

**Semantics.** Assuming a finite, non-empty set of \texttt{n} processes \texttt{p}, the state of a PSYNC program is represented by the tuple \((SU, r, msg, HO)\) where:

- \(SU \in \{\text{Snd, Updt}\}\) indicates whether the next operation is \texttt{send} or \texttt{update};
- \(s \in [P \rightarrow V \rightarrow D]\) stores the local states of the processes;
- \(r \in \mathbb{N}\) is a counter for the number of executed round;
- \(msg \subseteq \mathbb{P}^P \times \mathbb{P} \times \mathbb{P}\) stores the messages in the system between the \texttt{send} and \texttt{update} phase of a round;
- \(HO \in [P \rightarrow 2^P]\) evaluates the HO-sets for the current round.

The semantics of a PSYNC program is shown in Figure 5.

A transition is written as \(S \xrightarrow{L,O} S'\) where \(S, S'\) are states, \(O\) is a set of labels from the interface, corresponding to observable transitions, \(L\) is a set of labels not in the interface corresponding to internal transitions. A client of a PSYNC program can only observe the transition labels in \(O\).

We consider the following shorthands: \(|\text{phase}|\) is the number of rounds in a phase and \(\text{phase}[r]\) is used to identify the \(r\)th round in a phase. For example, Fig. 1 declares \(\text{phase} = 4\) types of rounds and \(\text{phase}[3]\) identifies a Quorum round. The operation \(m\) of a round \(\text{phase}(r)\) is \(\text{phase}(r).m\). A transitions \(s(p) \xrightarrow{op,o} s'(p)\) says that \(p\) executes operation \(op\) in local state \(s(p)\) and reaches local state \(s'(p)\). The execution of \(op\) produces an observable transition \(o\), i.e., \(o\) is in the interface.

Initially the state of the system is undefined, denoted by \(*\), and the first transition of every process \(p\) is to call the \texttt{init} operation whose arguments are received from the client via an \texttt{init} transition executed by the process \(p\) (see Init in Fig. 5). The \texttt{init} operation does not return a value but initialize the state of the system. Initially, the round counter is 0, there are no messages in the system, and the first operation is \texttt{Snd}. An execution alternates \texttt{Send} and \texttt{Update} transitions from Fig 5.

During a \texttt{Send} transition, the messages sent by each process are added to the message buffer \(msg\). The messages in \(msg\) are triples of the form (sender, payload, recipient), where the sender and receiver are processes and the payload has type \texttt{T}. The \texttt{send} operation does not affect the state of the processes. The values of the HO-sets are defined non-deterministically by the environment.

In an \texttt{Update} step, messages are received and the \texttt{update} operation is applied locally on each process. The set of received messages is the input of \texttt{update}. A message is received only if the sender is in the receiver’s HO-set. The \texttt{update} operation might produce an observable transition \(o_p\). At the end of the round \(msg\) is purged and \(r\) is incremented by 1.

To obtain a transition system as in Section 3, the fields \(SU, r, HO\) are copied locally on each process, the interface along the \texttt{send} and \texttt{update} operations define the labels of the transition system and the pool of messages \(msg\) is represented by a special network process whose only local variable is \(msg\).

**Environment assumptions.** Many problems, such as consensus, are not solvable in asynchronous networks with faults. More precisely, the existence of an algorithm solving such a problem is predicated by assumptions on the network on top of which the algorithm is executed. Therefore, the algorithm designer must provide not only a PSYNC program but also the assumptions on the network its algorithm is designed for.

In PSYNC the network assumptions translate into assumptions on the environment actions. They are given as linear temporal logic (LTL) formulas over atomic propositions that constrain the values of the HO-sets. The classic taxonomy of distributed systems distinguishes the synchrony degree of the network, the reliability of the links, different types of process failure. The relation between the classic types of systems and the corresponding assumptions on the HO-sets is given in [1]. For example, if an algorithm is designed for asynchronous network with reliable links and at most \(f\) crash failures then in PSYNC the algorithm designer should assume an environment that assigns sets of cardinality greater than or equal to \(n - f\) to each HO-set, i.e., \(\forall p, |\text{HO}(p)| \geq n - f\), where \(n\) is the number of processes in the network.

We consider that network assumptions are required only to guarantee the desired liveness properties. Without making any assumption on the environment a program might never make any progress. For example, the environment can decide that the HO-sets are always empty and no message is delivered. In order to ensure termi-
nation LastVoting assumes that eventually there exists a sequence of four rounds, starting with Collect, where (1) the coordinator is in the HO-set of any process, and (2) during the Collect and Quorum rounds of this sequence the HO-set of each process contains at least \( n/2 \) processes. Formally, this environment assumption is expressed by the formula \( \left\langle \psi \land \phi \right\rangle \), where \( \psi \) is given in 1. Notice that at the end of these four rounds, all processes have decided the value proposed by the coordinator in the Collect round, even if no process has made a decision previously.

**Definition 4** (Execution). Given a PSYNC program \( P \) and a non-empty set of processes \( P \), an execution of \( P \) is the sequence \( *A_{0}A_{1}A_{2} \ldots \) such that:

1. \( *A_{0} \) is the result of the Init rule;
2. \( \forall i, A_{i}A_{i+1} \) satisfy the Send or the Update rule;
3. the environment assumptions on HO-sets are satisfied.

The set of executions of \( P \) is denoted by \( [P] \).

A lock-step execution of LastVoting has a finite prefix when the environment assumptions are met followed by a suffix that satisfies them. In general, lock-step executions of PSYNC programs are an alternation of bad and good rounds, where a round is good if the corresponding environment assumptions are met.

For any program \( P \), we consider that the environment assumptions of \( P \) are time-invariant, i.e., they are of the form \( \square \phi \), where \( \phi \) is the network assumption that \( P \) relies on during its execution. Time invariance is important because we don’t want the correctness of \( P \) to depend on the time it starts executing.

### 5. Runtime

In this section we define a runtime that executes PSYNC programs on a wide variety of network configurations. We start by describing the network assumptions, then the algorithm behind the runtime, and finally, we show that a client cannot distinguish between a PSYNC program and its runtime execution.

#### 5.1 Partial synchrony

The round structure can be accurately simulated on an asynchronous network provided that the latter satisfies the partial synchrony assumption. Partial synchrony means that the network alternates between bad and good time periods, where during a good period the communication and the processes are synchronous while through a bad period they are asynchronous. The good periods are needed to ensure progress of the system and the safety properties should hold during arbitrarily long asynchronous periods.

**Definition 5** (Synchronous network). A network is synchronous if there exists \( \Theta \) and \( \Delta \) two positive integers, such that:

1. \( \Theta \) is the minimal time interval in which any process is guaranteed to take a step;
2. \( \Delta \) is the maximal transmission delay between any processes.

Intuitively, \( \Theta \) corresponds to process synchrony and \( \Delta \) is communication synchrony. Both are required in order to solve problems like consensus [24].

When the network is asynchronous there are no bounds on the communication delay and relative speed.

**Remark 1.** The transmission delay is typically much larger than the time required to ensure a computation step, i.e., \( \Delta \gg \Theta > 0 \).

**Definition 6** (Partially synchronous network [31]). A network is partially synchronous if the constants \( \Theta, \Delta \) exist, are known, and eventually hold after a time \( \text{gst} \), called the global stabilization time.

---

1 In [31] a second definition is proposed where \( \Theta \) and \( \Delta \) are unknown but they hold from the beginning. Our results hold under both definitions of partial synchrony. We have chosen Def. 6 for performance reason.

Def. 6 characterizes the partial synchrony assumption w.r.t. an initial time. We assume this time coincide with the start of the program execution. To avoid depending on the time processes start, a network is partially synchronous if it alternates between good and bad time periods, where the good periods are as long as needed. The required length of good periods depends on the program. Def. 6 takes the supremum of good periods for any program to allow a general statement about liveness of the system.

#### 5.2 Runtime Algorithm

The runtime system is defined by the asynchronous composition of all processes in the network intersected with the partial synchrony assumptions. Given a PSYNC program \( P \), Fig. 6 shows the code executed by each process to run \( P \). Roughly, processes execute locally the same sequence of rounds as in \( P \), but the parallel composition is asynchronous.

The algorithm in Fig. 6 uses two while-loops. One iteration of the outer loop (line 9) executes one round. The inner loop (line 15) accumulates messages until a timeout is reached. Because the network is not synchronous, the runtime deals with messages of past or future rounds, i.e., messages tagged with round numbers strictly smaller or bigger than the process’s current round number. Late messages are dropped (line 18), and a message from a future round forces the runtime to execute the outer loop until it catches up and reaches the round of the received message (line 11). The send and update operations, on line 10 and 26, are those defined in the executed PSYNC program \( P \). To deal with messages duplication the accumulated messages are stored in a set. The function tryReceive(d) tries to receive a message if one is available, or becomes available during the next \( d \) time units. If no message is available in this period then the method returns 1.

The variable to has the same reference value across processes. It is used to measure locally a time interval of length \( to \) in reference time units. The function currentTime returns the value of a local clock, s.t., all processes measures the same duration for a time interval of length \( to \). We assume that the processor’s speed is not related to its clock and also we assume a bounded clock drift across processes. Therefore, processes can execute a different number of instructions while measuring the same interval.

Messages are tuples of the form \((\text{sender}, \text{payload}, \text{receiver}, \text{round})\), where \( \text{sender}, \text{receiver} \) are the respective sender and receiver of the message, \( \text{payload} \) is the content of the message, and \( \text{round} \) is the sender’s round number when the message was sent.

In the following, given a PSYNC program \( P \), we define the semantics of the runtime of \( P \). A state of the Runtime of \( P \) is represented by the tuple \((s \upharpoonright s_{v}, \text{msg})\) where:

1. \( s \in [P \rightarrow V \rightarrow D] \) is an evaluation of the variables in \( P \);
2. \( s_{v} \in [P \rightarrow V \rightarrow D] \) is an evaluation of the variables introduced by the runtime;
3. \( \text{msg} \in [(P, T, P, N) \rightarrow N] \) is a multiset of messages in transit.

In Fig. 7 we define the semantics of the most important instructions of the algorithm in Fig. 6. Each transition has the form \( s \xrightarrow{\text{TO}} s' \), where \( s, s' \) are global states and \( I, \text{res.} O, \text{res.} \) are the internal, resp. observable labels of the transition. The runtime of a program \( P \) interacts with a client using the interface of \( P \) defining the observable transitions. By considering \( \text{msg} \) a specific network process, we obtain a transition system as defined in Section 3.

**Local process transitions.** The Send rule states that the messages sent by one process are tagged with the current round of the sender, i.e., \( s(p).r \), and added to the global pool of messages, \( \text{msg} \). The messages sent by one process in round \( r \) are defined by the send operation of the same round from \( P \). The \text{RECEIVE1} and \text{RECEIVE2} rules define the reception of a message. The \text{RECEIVE2}
The system of transitions defined in Fig. 7 generates purely asynchronous executions and some of them are not possible under the partial synchrony assumption. Therefore we consider an execution π = s₀,A₀,s₁,A₂,... of the transition system in Fig. 7 satisfies the partially synchronous assumption on the network iff there exits a global stabilization time, gst, s.t. γ = πₐπₕ and

- for any s ∈ πₐ, γ(s) > gst;
- for any i, j such that γ(sᵢ) > γ(sⱼ) and γ(sⱼ) ≥ GST, every process takes at least one step the sequence sᵢ...sⱼ;
- for any i,j such that γ(sᵢ) > γ(sⱼ), sendₐ(mᵢ) ∈ Aᵢ, and receive(mⱼ) ∈ Aⱼ, with m ∈ ms, either the delay is 0 < γ(sᵢ) − γ(sⱼ) ≤ Δ or let k ∈ [i,j] the first state with γ(sᵢ) − γ(sⱼ) > Δ then q does not take any WAIT or RECEIVE2 step in the sᵢ...sⱼ interval;
- there is no message duplication after γ(sⱼ).

Limiting duplication only simplifies the proof. It is possible to tolerate a bounded amount of duplication after γ(sⱼ). The time to process the duplicate messages has to be factored in the timeout.

### Local transitions

- **SEND**
  
  \[ p \xrightarrow{send(mᵢ)} p' \]

- **RECEIVE1**
  
  \[ p \xrightarrow{receive(mᵢ)} p' \]

- **RECEIVE2**
  
  \[ p \xrightarrow{receive(_⊥)} p' \]

- **UPDATE**
  
  \[ p, m \xrightarrow{update(mᵢ)} p' \]

- **DROP**
  
  \[ p, ms \xrightarrow{drop(mᵢ)} p, ms' \]

- **CRASH**
  
  \[ p \xrightarrow{crash(sᵢ)} p' \]

- **DUPLICATE**
  
  \[ p, ms \xrightarrow{duplicate(mᵢ)} p, ms' \]

### Global transitions

- **INIT**
  
  \[ \forall p \in P. \xrightarrow{\text{init}(p)} s(p) \]

- **PARALLEL COMPOSITION**
  
  \[ P' \subseteq P \]

  \[ ms = \bigcup_{p \in P} ms_p \]

  \[ ms' = \bigcup_{p \in P'} ms'_p \]

  \[ \forall p \in P' \]

  \[ s(p), ms \xrightarrow{a \in A} s'(p), ms' \]

### Figure 6: Algorithm to implement the round structure

```plaintext
// Local variables
p // initialized PSync process
to // timeout
r := 0 // current round number
mbox := \emptyset // last received message
msg := \emptyset // messages received but not yet processed
t := currentTime() // time at which the current round began

while (true) {
    p.phase[r % p.phase.size].send() // send event
    if (msg.round = r) {
        msg := \emptyset
    } else if (msg.round < r) {
        msg := \emptyset
    } else if (msg.round > r) {
        msg := \emptyset
    }
    p.phase[r % p.phase.size].update(mbox) // update event
    if (msg) {
        msg := \emptyset
    } else if (msg = \emptyset) {
        msg := \emptyset
    } else if (msg.round = r) {
        msg := \emptyset
    } else if (msg.round < r) {
        msg := \emptyset
    } else if (msg.round > r) {
        msg := \emptyset
    }
    p.phase[r % p.phase.size].update(mbox) // update event
    r := r + 1
    t := currentTime()
    mbox := \emptyset
}
```

### Figure 7: Runtime semantics of a PSync program P

- **INIT**
  
  \[ \forall p \in P. \xrightarrow{\text{init}(p)} s(p) \]

- **PARALLEL COMPOSITION**
  
  \[ I = \bigcup_{p \in P} I_p \]

  \[ O = \bigcup_{p \in P} O_p \]

  \[ \forall p \in P' \]

  \[ s(p), ms \xrightarrow{a \in A} s'(p), ms' \]

### Definition 7 (Executions of the runtime system)

We define the set of executions of the runtime system associated with a PSync program P, denoted \( \mathcal{E}(\mathcal{P}) \), by the set of partially synchronous executions of the transitions system in Fig. 7 which satisfy the environment assumptions defined in P.

**Remark 2.** The environment assumptions are time invariant, i.e., they are LTL formulas of the form \( \Box \phi \). Therefore we consider \( \pi = \piₐπₕ ∈ \mathcal{E}(\mathcal{P}) \) if \( \piₐ \) satisfies the environment assumption if \( \piₕ \) satisfies \( \Box \phi \).

The runtime does not include HO-sets. To evaluate the environment assumptions from P on the runtime executions, HO-sets are replaced with the set of received messages. That is, any constraint on \( \text{HO}(p) \) in round r, is replaced by the same constraint over
mbox_p, the set of messages received by process p in round r. The update called by process p in round r takes mbox_p as argument.

In the following we identify values for the timeout variable which ensure that any execution of the runtime implements correctly the round structure under the partially synchrony assumption allowing the reception of all the messages that can be received in a round. We begin with the value of the timeout when the network is synchronous and then we compute the timeout for partially synchronous networks. We assume that every process periodically communicates with every other process. If the algorithm does not send messages, the runtime inserts heartbeat messages. Communication is needed for synchronization.

Definition 8 (Synchronous execution). An execution of $[P]_{rt}$ is called synchronous if in any time interval of length $\Theta$ each process takes at least a send update, or receive step and every message sent in a given round is received in the same round, dropped by the network, or the receiver has crashed.

Lemma 1. If the network is synchronous any execution of $[P]_{rt}$ is synchronous if the value of the timeout is $(n+1)\Theta + \Delta$.

When the network is synchronous the maximal duration of a round is given by the slowest process and it equals $(n+1)\Theta + \Delta$. The runtime will force the slowest process to directly call the update and move to the next round, taking $2\Theta$. That message will be at most $3\Theta + \Delta$ behind the fastest process. Therefore, the fastest process will receives the messages from the slowest process at most $4\Theta + 2\Delta$ after is started the round. As their might be $(n-1)$ such messages, the timeout needs to be at least $(n+3)\Theta + 2\Delta$ to receive the remaining messages for the current round. Fig. 8 illustrate how the synchronizations happens.

After the system reaches synchrony. The remaining suffix $\pi_s$ stays synchronous, the timeout is greater than the sum of $(n+1)\Theta + \Delta$ (timeout for the synchronous case) + $3\Theta + \Delta$ (maximal slack) - 1 (the message send by a process to itself can be processed more quickly).

5.3 Correctness

Theorem 2. For any PSYNC program $P$, the transition system associated with the runtime of $P$ is indistinguishable from the transition system associated with $P$, i.e., $\forall P, [P]_{rt} \equiv [P]_{r'}$.

Proof. (Sketch) Given an execution $\pi$ from $[P]_{rt}$, we need to build an execution $\pi'$ such that $\pi' \equiv [P]$ and $\pi \equiv [P]_{rt}$. We recall that the local variables $V$ declared in $P$ are modified only by the update operation, which has the same semantics in both $[P]$ and $[P]_{rt}$. Therefore, in order to build $\pi'$ we need to show that (1) we can associate to every round $r$ of $\pi$ an execution of the same round under the PSYNC semantics, with the same output for the send operation and the same input for the update operation and (2) this sequence of PSYNC rounds satisfies the environment assumptions.

The execution $\pi'$ is build from $\pi$ by eliminating all receive steps, and shifting send and update operations to the left and right such that they execute in lockstep. Shifting the send and update of a process preserves the order of the operations within that process. For each round $r$ the environment defines HO(p) to be the identity of the senders of the set of messages delivered to $p$ tagged by round $r$. Without loss of generality, in $[P]_{rt}$ we assume that at each round every process sends a message to every other process. If the algorithm sends less messages we can introduce additional ghost messages for the proof purpose.

By Lemma 2, $\pi = \pi_a \pi_s$, where the second part is synchronous. Due to the time invariance of the environment assumptions and that $\pi_s$ is finite, we can find a corresponding $\pi'_s$, a prefix of $\pi'$, where all the messages are delivered. In $\pi_s$, all the messages are either delivered or dropped so $\pi_s$ respects the environment assumptions and the corresponding $\pi'_s$, with $\pi' = \pi'_a \pi'_s$, that also satisfies them.

Finally, the crashed processes, represented by _ in $\pi$, can be matched to any lockstep execution where the corresponding processes do not appear in the HO set of any process after the crash.

The execution $\pi$ is indistinguishable from $\pi'$ because (1) for every round the state preceding the send, resp. update, operations is the same in $\pi$ and $\pi'$; (2) for every round the update operations have the same inputs in both $\pi$ and $\pi'$; (3) receive steps in the runtime trace correspond to stuttering in the lockstep trace.

Corollary 1. For any PSYNC program $P$, $[P]_{rt} \equiv [P]$.

6. Verification

In this section we identify the class of specifications such that if a PSYNC program satisfies the specification then its runtime system satisfies it as well. We underline the advantages of PSYNC for automated verification. Finally, we present a deductive verification engine for PSYNC.

6.1 From verified PSYNC to verified runtime executions

We consider specifications given as sets of sequences of program states, or sets of runs. A specification $Spec$ is a set of sequence of states in $\Sigma = [P \rightarrow V_p \rightarrow D]$, where $V_{spec} = \bigcup_{p \in P} V_p$. The specification does not talk about the HO-sets. A spec to be applicable to a PSYNC program with the set of variables $V$, if
Consensus correct processes eventually decide. These properties correspond to (1) Capability, a process cannot change its decision; (2) Validity, the decision is the initial value of a process; (3) Irreversibility, a process cannot change its decision; (4) Termination, all processes stop after some finite time executions. For clients interacting with a PSYNC system $(P, V, A, s_0, T)$, the last voting decision $s(t)$ among processes in $V$ satisfies

$\forall s \in A, (s, b) \in T, a \in B \Rightarrow s \notin s(p) | W_p \notin f(a) \land s(p) | W_p \in f(a)$.

The input-output interface $\text{LastVoting}$ w.r.t Consensus, called $(I_c, f_c)$, associates $\text{init}(p)(v_p)$ with the first initialized state when $s(p).x = v$ and $\text{out}(v_p)$ when decided is set to true and decision is $v$ for every $p \in P$. The mapping $f_c$ relates $(\text{init}(v), v)$ to $\{s \mid s(p).x = v \land s(p).ts = 1\}$ and $\text{out}(v)$ to $\{s \mid s(p).decided \land s(p).decision = v\}$. A PSYNC program might require the addition of variables to ensure that input-output interfaces exist. For instance, the decision variable in $\text{LastVoting}$ changes values when the out event occurs.

Given a specification Spec and an input-output interface $(I, f)$, $TS(Spec, f)$ is the transition system obtained from $TS(Spec)$ by setting $A = I$ and changing $T$ to add labels according to Def. 9.

If an interface is compatible with a program and a specification then it relates the program’s executions to the executions of the transition system obtained from the specification.

**Proposition 1.** For any PSYNC program $P$, a specification $Spec$, and an input-output interface $(I, f)$ for $P$ and $Spec$:

$$\text{Runs}(P)_{V_{Spec} \subseteq Spec} \Rightarrow [P]_{V_{Spec}, I} \subseteq \{TS(Spec, f)\}_{V_{Spec}, I}$$

To relate a specification to the runtime executions, we need to account for indistinguishability. Given a transition system $TS$ we denote by $\text{Closure}_\equiv (TS)$ its closure w.r.t. the indistinguishability relation, i.e., $\{\text{Closure}_\equiv (TS)\} = \{s \mid 3s. \pi \equiv \pi'\}$. 

**Theorem 3.** For any PSYNC program $P$ with an input-output interface $(I, f)$, and specification $Spec$, if $P$ satisfies $Spec$ and $TS(Spec)$ is closed under indistinguishability, then $[P]_{rt} \supseteq \{TS(Spec, f)\} \cap [P]_{rt} \subseteq I\{TS(Spec, f)\}$. 

**Proof.** Let $\pi \in [P]_{rt}$ be an execution of the runtime of $P$. Th. 2 implies that there exists $\pi' \in [P]$ s.t. $\pi \equiv \pi'$. Moreover, since $V_{Spec}$ is included in the set of program variables in $P$ and the interface $I$ is common for $P$ and its runtime, (and indistinguishability between the runtime and PSYNC holds w.r.t. all the variables in $P$ and all common labels), it implies that $\pi|_{V_{Spec}} \equiv \pi'|_{V_{Spec}}$.

Since $P$ satisfies the specification, from Prop. 1 it follows that $\pi'|_{V_{Spec}} \in \{TS(Spec, f)\}$. Finally, since $\{TS(Spec, f)\}$ is closed under indistinguishability $\pi|_{V_{Spec}} \in \{TS(Spec, f)\}$. Therefore, $[P]_{rt} \supseteq \{TS(Spec, f)\}$ and, using Th. 1, it follows that $[P]_{rt} \subseteq I\{TS(Spec, f)\}$. $\square$

**Proposition 2.** Consensus is closed under indistinguishability and if a program $P$ satisfies Consensus then $[P]_{rt} \subseteq I\{P\} \subseteq I\{TS(Consensus, f_c)\}$. 

Roughly, consensus is closed under indistinguishability because it does not impose an order on the updates performed on different processes: processes can decide on a value in any order and at any time. The same reasoning also holds for other agreement specifications, like the k-set agreement [26], or the lattice agreement [33].

**6.2 Benefits of PSYNC for verification**

Distributed algorithms are challenging to verify because of several sources of unboundedness. Messages come from unbounded domains, the number of processes is a parameter, and channels may also be unbounded. Using communication-closed rounds and a lockstep semantics helps mitigate or avoid these challenges.

**Model checking** Model checking techniques are based on algorithms that explore the system’s reachable states. It requires a fixed number of finite state processes. With an asynchronous semantics, a model checker explores all the possible interleavings of processes transitions and suffers from combinatorial explosion. In the lockstep semantics of PSYNC all the processes take a step at the same time removing the interleavings. Another difficulty comes from the communication channels. Unbounded FIFO channels causes undecidability even for two processes [17]. Making the channels lossy [3] and fixing the number of processes makes the problem non-primitive recursive [64]. Weaker channel models are usually at least EXPSPACE-hard for verification. Communication-closed rounds sidestep this difficulty.

**Deductive verification** Deductive verification relies on user provided inductive invariants and ranking functions. The invariants describe an over-approximation of the set of reachable states which is inductive w.r.t. the program transitions. Ranking functions show progress toward satisfying the program goals. However, finding these annotations is not easy even for an expert. Automated techniques, such as static analysis, are far from being able to generate these annotations automatically for our targeted class of systems.

The lockstep semantics leads to much simpler invariants, because they are required to describe the set of reachable states only at the boundaries between rounds.

In the literature, the HO-model has been shown to be suited for verification using bounded state-space exploration [22, 66–68] and interactive theorem provers [23, 25, 27, 54].
6.3 A verifier for PSYNC

We consider specifications that include both safety and eventuality properties and are formally defined in LTL with state properties in the logic $\mathsf{CL}$ [30]. The verifier inputs are the specification and a PSYNC program annotated with inductive invariant candidates. The verifier checks the validity of the invariants and that they imply the specification by generating verification conditions that can be discharged using an SMT solver.

Expressiveness $\mathsf{CL}$ is a first-order logic over sets of program states. The variables 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 $x(p)$, where $x$ is a function of type $P \to T$ with $P$ is the type of processes. To characterize global states, $\mathsf{CL}$ uses universal quantification over variables of type $P$, set comprehensions, and cardinality constraints.

The programmer provides the inductive invariant candidates and the pre/post condition of the $\text{send}$, $\text{update}$ functions. Typically the correctness argument for consensus solving algorithms, which an invariant must capture, is centred around the existence of majority of processes that agree on value $v$, $\exists v, \{p \mid x(p) = v\} > n/2$ defines a majority ($> n/2$) of processes that agree on value $v$ using a comprehension, where $x$ is the function symbol associated with the local variable $x$.

The inductive invariant that shows agreement in $\text{LastVoting}$ is

$$\forall p. \neg \text{decided}(p) \land \neg \text{ready}(p) \lor \exists r, t, A. A = \{p \mid t_\text{sn}(p) \geq t\} \land t < r/4 \land |A| > n/2 \land \forall p, p \in A \Rightarrow x(p) = v \land \forall p. \text{decided}(p) \Rightarrow \text{decision}(p) = v \land \forall p. \text{commit}(p) \lor \text{ready}(p) \Rightarrow \text{vote}(p) = v \land \forall p. t_\text{sn}(p) = r/4 \Rightarrow \text{commit}(\text{coord}(r/4)).$$

The invariant is a case split characterizing the states in which processes can safely decide. A process decides when there is a majority of processes agreeing on a proposal with timestamps more recent than $t$. The additional clauses are required to make the invariant inductive and to relate it to the specification. For instance, if a process is ready then its vote is $v$ and it agrees with the majority. Also any process that has decided its decision is $v$.

Methodology To prove safety properties we implement a standard verification conditions generator for PSYNC programs. For the round $R$, the generator builds a $\mathsf{CL}$ formula corresponding to the transitions relation as follows. Let $s, s'$ be the primed and unprimed function used to represent the global state of the system. The transition relation associated with a local $\text{send}$, resp. $\text{update}$, of round $R$ is $\text{send}_{R}(s(p), m)$, resp. $\text{update}_{R}(m, s(p), s'(p))$.

Then the transition relation of $R$, $\text{TR}_{R}(s, s')$ is

$$\forall p. \text{send}_{R}(s(p), m_{s}(p)) \land \forall q, t. (t, q) \in m_{s}(p) \Rightarrow (t, p) \in m_{s}(q) \land q \in \text{HO}(p) \land \forall p. \text{update}_{R}(m_{s}(p), s(p), s'(p)) \land r' = r + 1.$$  

Safety We generate verification conditions that imply partial correctness: (1) the invariant contains the initial state ($\text{TR}_{\text{init}}(s) \Rightarrow \text{Inv}(s)$), (2) for any round $R$ the invariant is inductive ($\text{Inv}(s) \land \text{TR}_{R}(s, s') \Rightarrow \text{Inv}(s')$), (3) the safety specification $\varphi$ is implied by the invariant ($\text{Inv}(s) \Rightarrow \varphi(s)$).

Eventuality We also prove eventuality properties, such as, every process eventually decides in $\text{LastVoting}$. Showing these properties typically requires ranking functions. However, in many cases we can simplify the proof. We show that there exists a fixed number of good rounds, i.e., rounds when the environment assumptions hold, such that after the execution of the last good round the process reaches a set of good states, e.g., processes have decided. To prove that the program makes progress after each good round the user provides additional invariants, expressing how a good round strengthens the safety invariant. For instance, in the $\text{LastVoting}$ the program makes a decision if the formula (1) holds during one complete phase of the algorithms. An intermediate invariants specify that between round Collect and Candidate the formula $\text{commit}(\text{coord}(r))$ holds on top of the safety invariant.

7. Evaluation

We have implemented PSYNC as an embedding in the SCALA programming language. The runtime of PSYNC is built on top of the NETTY [2] framework. For the transport layer, we use UDP. The serialization of messages uses the pickling library [56].

The set of replicas executing a PSYNC program is specified in a configuration file. The runtime manages the interface between a PSYNC program $P$ and the client application, and also the communication between the different replicas running $P$.

The execution of $P$ is launched from a client application, using $P$’s interface. More specifically, the interface contains callback methods provided by the application. Regarding termination, many consensus algorithms presented in the literature assume that processes continue executing the algorithm after a decision is taken, because not all processes decide simultaneously. For example, in a case of a network partition, some processes learn the decision value much later. To safely free the memory allocated by the runtime when a process decides, each replica stores only the decision value in a log. In our experiments we implement a key-value store using $\text{LastVoting}$ iteratively. On each replica we terminate an execution of $\text{LastVoting}$ after a process decides and keep a log of the most recent decisions. Replicas which have terminated executing $\text{LastVoting}$ detect messages from the late replicas and send them the decision.

To achieve the good performances, the timeout is an important parameter determined empirically. $\Delta$ can easily be measured, i.e., latency and bandwidth. $\Theta$ is harder to measure but can easily be over-approximated. To decrease the reliance on an accurate timeout, we implemented several optimizations that allows the runtime to progress before the timeout occurs.

7.1 Implementing Algorithms in PSYNC

PSYNC can implement a wide variety of fault-tolerant distributed algorithms. Table 1 lists several implementations in PSYNC of algorithms that solve different agreement problems. For each algorithm, we indicate if it is designed for a synchronous or asynchronous network and if the presentation uses some form of rounds. Many asynchronous algorithms are tagging messages with information that implicitly structures programs in rounds (not necessarily executing in lockstep). However, even when the original algorithm presentation is event-driven, they can be encoded in PSYNC.

The first three algorithms focus on the traditional consensus problem, the others are weaker agreement problem. Ben-Or algorithm [13] solves binary consensus and almost surely terminates. The k-set agreement [26] is a weaker version of consensus that allows processes to decide on k-different values. The generalized
Table 2: Comparison of the code size and verification of Paxos in different languages. For EventML, Schiper et al. [63] report the number of AST nodes. The sources locations are in Appendix C.

We compare PSYNC against other high-level languages for distributed algorithms. Table 2 shows a comparison between different implementations of Paxos in different programming and specification languages. For PSYNC we used LastVoting. For each implementation we count the number of lines of code without comments or blank lines. Also we focus on the algorithm itself and remove boilerplates like include statements.

We compare against the following languages: DistAlgo [50] is a programming language for distributed algorithm that uses incrementalization to compile a high-level specification into Python code. Distal [16] is designed to express fault-tolerant distributed algorithms in a pseudo-code-like manner. It is built as a library on top of Scala. Bloom [7] is a programming language based on a monotonic logic for building consistent distributed systems. Overlog [6] is a logic programming language for distributed systems inspired by Datalog. EventML [55] is a programming and verification language for distributed algorithms connected to the Nuprl interactive theorem prover [5]. Verdi [70] is a Coq framework for implementing and proving distributed systems correct. TLA+ [48] is a logic-based specification language designed to describe concurrent and distributed systems. IO Automata [52] is a specification language with automata theoretic foundations to describe asynchronous concurrent and distributed systems. Except for TLA+ which can encode both synchronous and asynchronous programs, and Verdi that starts with a synchronous model and transforms it into an asynchronous one, the other languages have an asynchronous semantics. Currently only mechanized proofs in Nuprl or Coq exist for Paxos, when implemented in EventML or Verdi.

**Limitations of the model** PSYNC keeps an order between messages only if they are sent in different rounds. It is oblivious to the order in which messages arrive in one round. As a consequence, one cannot implement the runtime system of PSYNC in PSYNC itself. Also, for the moment we don’t have an efficient way of composing PSYNC programs to create other programs. The composing round based models is a problem that currently receives attention.

### 7.2 Comparing PSYNC to existing Paxos implementations

We evaluate our PSYNC implementation of Paxos (LastVoting from Fig. 1) and compare it to existing Paxos implementations. We use LastVoting to order write requests in a simple key-value store. The submitted requests are collected into batches of about 300 requests, then LastVoting is used to make all replicas agree on the next set of writes. Adding batching to LastVoting requires only changing the type of the messages sent in each round, i.e., modifying only a few lines of code (<10) because batching does not interfere with the control structure of the algorithm.

Table 3 shows throughput numbers for the different implementations of Paxos we considered. All the algorithms try to maximize the throughput when dealing with requests of small sizes. However, the exact settings is slightly different for every experiment. Due to the difficulty of replicating published results, we report the published numbers. We run PSYNC on three servers with Intel Xeon X5460 cpu and 8 GB ram, running Linux 2.6.32 and the JRE 1.8. We also ran experiments incorporating crashes. The throughput of the system roughly halves after one crash. The point of this comparison is to show that the overhead of the runtime to implement the round structure is acceptable and does not preclude PSYNC adoption. We believe that the benefits of PSYNC, i.e., an intuitive semantics, simple control structures, and the ability to use automated verification tools, make it a compelling language.

PSYNC and Distal both are based on CALA. JPaxos and SPaxos [15] are written in Java. JPaxos is Java implementation of Paxos and SPaxos is an improved algorithm to achieve higher-throughput when the coordinator is CPU bound. Egalitarian Paxos [59] is implemented in Go and improves over Paxos by processing independent requests in parallel. Paxos for system builder [8] is an implementation of Paxos in C. To achieve high throughput, all the implementations use batching.

#### 7.3 Verification in PSYNC

We implemented a verification conditions generator for PSYNC, based on the logic CL. To compute the verification conditions the tool computes, at compile time, (1) the transition relation of the program and (2) transforms the program assertions given in the CALA language in CL formulas.

We implemented the semi-decision procedure for CL [30] on top of the SMT solver Z3 [60] and using the VC generator we verified the PSYNC programs One third rule [24] and LastVoting. Their specification and invariants are provided by the user. We used the invariants from [30]. For the One Third Rule, we need for 4 invariants (23 LOCS), 27 VCs are generated, solved in 5s. For the LastVoting, we need for 8 invariants (35 LOCS), 45 VCs are generated, solved in 16s.

The verification of programs solving weaker agreement problem requires reasoning about sets of data. For example, k-set consensus needs to reason about the cardinality of the set of decision values. CL supports only reasoning about sets of processes. We are working on extending the scope of our implementation to verify a larger class of examples.

#### 8. Related Work

In this section we compare PSYNC to the related work from a programming language and verification perspective.

**Formalizations of distributed algorithms** Distributed algorithms are typically defined in English or pseudo-code [39] using different

---

Table 3: Performance of Paxos implementations with 3 replicas

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Source</th>
<th>Year</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastBiding (Batching)</td>
<td>2015</td>
<td>✓</td>
<td>170</td>
</tr>
<tr>
<td>Egalitarian Paxos</td>
<td>[59]</td>
<td>2013</td>
<td>450</td>
</tr>
<tr>
<td>Paxos in Distal</td>
<td>[16]</td>
<td>2013</td>
<td>150</td>
</tr>
<tr>
<td>Paxos for system builder</td>
<td>[8]</td>
<td>2008</td>
<td>40</td>
</tr>
</tbody>
</table>

1 We use 5 years old machines which makes the comparison with older results relevant.
computational models. Synchronous and asynchronous models are the most frequent ones. Synchrony allows solving a larger class of problems, while asynchrony is close to the network behavior. Finding a uniform model is still an open problem in the distributed algorithms community. Multiple models that abstract uniformly faults and (asynchronous have been introduced [4, 14, 24, 31, 37, 62, 69]. We have chosen the HO-model [24] because of its simplicity. It handles asynchrony, host and network failures uniformly.

In the classic setting, distributed systems are formalized using the π-calculus [57, 58], CSP [42], and I/O-automata [52]. These formalisms are used to give a formal semantics to message-passing systems and to analyze them, but not as programming languages.

The Actor model [41] is probably the most successful high-level programming abstraction for message-passing systems. Actors are either built-in languages, e.g., Erlang [10], or supported through libraries, e.g., Scala [40]. Erlang via the OTP library [65] has support to handle faults in distributed systems. Faults are handled using a supervision hierarchy: when a replica fails its superior in either built-in languages, e.g., Erlang [10], or supported through programming abstraction for message-passing systems. Actors are formalisms are used to give a formal semantics to message-passing π-calculus [57, 58], CSP [42], and I/O-automata [52]. These formalisms are used to give a formal semantics to message-passing systems and to analyze them, but not as programming languages.

Several domain specific languages for distributed algorithms have been developed [6, 7, 11, 16, 45, 50, 55]. Languages like Meld [11], Overlog [6], and Bloom [7], which are based on DataLog, do not have a formal operational semantics, and do not support automated verification. The closest domain specific languages to PSYNC are Mace [45], DistAlgo [50] and Distal [16]. However, they all have an asynchronous semantics and lack high-level programming construct to reason explicitly about faults.

**Verification** The verification of parametric systems is in general undecidable [9]. Therefore, mainly bug finding tools are developed. They are based on state-space exploration up to a bounded, generally small, number of processes or messages [29, 36, 45, 66, 67], or uses specialized abstractions [43]. Static analysis techniques [1] are used to prove only simple properties, such as type errors or dead code detection, not for complex functional properties.

Recently a few programming languages were designed with build-in support for verification. Mace [45] has an integrated model checker which cannot prove total functional correctness. The most successful programming languages from a verification perspective are EventML [55] and Verdi [70]. EventML is a functional programming language and Verdi is a Coq framework for implementing and proving correct distributed algorithms. Verdi starts with a synchronous implementation and progressively transforms it using refinement into an asynchronous fault-tolerant one. The correctness of the implemented programs is mechanically proved using theorem provers: EventML uses Nuprl and Verdi uses Coq. These two languages have a small trusted base but reduced performances. For example, the throughputs of PSYNC implementations are several orders of magnitude faster than corresponding ones in Verdi. Moreover PSYNC’s verifier is designed for automated verification, currently based on SMT solvers.

In the algorithms community specification languages like +Cal [49] and TLA+ [48] are used to write formal specification of distributed algorithm and to model check or to mechanically prove them. However, these specifications are not executable.

Finally, the exploration of synchronous behavior of asynchronous systems for verification purposes has been investigated in [12, 28]. Our approach starts with a (partially) synchronous abstraction rather than retrofitting synchrony in existing asynchronous systems. The approach in [12, 28] makes the verification more complex without offering any guarantees about its applicability.

9. **Conclusion** We have presented PSYNC a domain specific language for fault-tolerant systems that strikes a balance between high-level constructs, performance, and automated verification. PSYNC offers a simple lockstep semantics that is indistinguishable from its runtime asynchronous executions. We have implemented a prototype runtime for PSYNC for partial synchronous networks and shown that it performs within a constant factor from highly optimized low-level implementations. For future work we intend to enlarge the application domain of PSYNC and to raise the automation level of the verification engine by developing static analysis that generate inductive invariants. We plan to generalize the runtime to cover more fault-models, such as Byzantine faults.

**Acknowledgments** This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award). Yours XXX I think we should thank Josef and while at it the reviews. Josef didn’t contribute to the paper but the initial discussions with him were useful.

**References**


A. Applicability to network assumptions

In the above, we focus on the case of partially synchronous system where crashed processes do not recover. However, PSYNC is not restricted this assumption and the runtime can be adapted for a wider variety of systems. We now give an overview of how to implement a round structure under different assumptions.

Alternative definition of partial synchrony. The runtime algorithm also works with the definition of partial synchrony where the bounds are not known. In this setting, the runtime must use incremental timeout, e.g., on line 19 in Fig. 6. A more realistic model allows the alternation of good and bad periods. Our system can works also the runtime must store in stable memory (hard disk) the state change to processing messages with the lowest round numbers first.

Crash-recovery. Crash-recovery is also possible. To allow recovery the runtime must store in stable memory (hard disk) the state of the system before sending messages. Upon recovery, the system resumes from the last state in the stable memory.

Byzantine failures. Byzantine failure requires a more complex algorithm to implement the round structure. First, the messages needs to be signed (HMAC) to avoid one process trying to impersonate another. Also the moving to the next round requires receipt of enough messages (2/3 majority). Otherwise, a Byzantine process can send messages with higher and higher round number and keep the system in an eternal catching-up game.

B. Additional definitions

Definition 10 (Distributed client). Let $TS_i = (\{p_i\}, V_i, A_i, s_{i0}, T_i)$ be the transition system associated with a client process, with $A_1 \cap A_i = \emptyset$ for all $1 \leq i \neq q \leq n$. Formally, the transitions system associated with the client is $C_{TS} = (P, V, A, s_0, T)$, where $P = \{p_1, p_2, \ldots, p_n\}, V = \bigcup_i V_i, A = \bigcup_i A_i, s_0 = (s_{01}, \ldots, s_{0n}),$ and $T \subseteq \Sigma \times A \times \Sigma, with \Sigma = [P \rightarrow V \rightarrow D], such that \Sigma(p_i) \in [V_i \rightarrow D], and (s, B, s') \in T iff for every $b \in B \cap A_i, (s(p_i), b, s'(p_i)) \in T_i$ and each processes takes at most one transition.

Definition 11 (Synchronized product). Let $TS_1$ and $TS_2$ be two labeled transition systems such that $I \subseteq A_2$ is an interface of $TS_2$ and $I \subseteq A_1$. We defined the synchronized product of $TS_1$ and $TS_2$ with respect to $I$, denoted $TS_1 \times_I TS_2$, as the asynchronous product of transition systems where the transition labeled in $I$ are synchronized. Formally, $TS_1 \times_I TS_2 = (P, V, s_0, T, A)$, where $P = P_1 \cup P_2, V = V_1 \cup V_2, s_0 = (s_{01}, s_{02}), \Sigma = \Sigma_1 \times \Sigma_2, A = A_1 \cup A_2$, and for any $s, s' \in \Sigma_1, t, t' \in \Sigma_2, for any a \in A_1 \setminus I, if (s, a, s') \in T_1 then ((s, t), a, (s', t)) \in T; for any a \in A_2 \setminus I, if (t, a, t') \in T_2 then ((s, t), a, (s, t')) \in T, and for any a \in I, if (s, a, s') \in T_1 and (t, a, t') \in T_2 then ((s, t)a(s', t')) \in T.$

C. Sources of for the programs size comparaison


Bloom https://github.com/bloom-lang/bud-sandbox/tree/master/paxos

Overlog https://bitbucket.org/neilconway/overlog-paxos/src/tip/src/olg/core/election.olg

IO Automaota in the IO-Automaota toolkit http://groups.csail.mit.edu/tds/ioa/

TLA+ https://github.com/fintler/tlaplus/blob/master/examples/Paxos/Paxos.tla

Verdi we took the number reported in [70].

EventML we took the number reported in [63].

These references were accessed on 21st of March 2015.