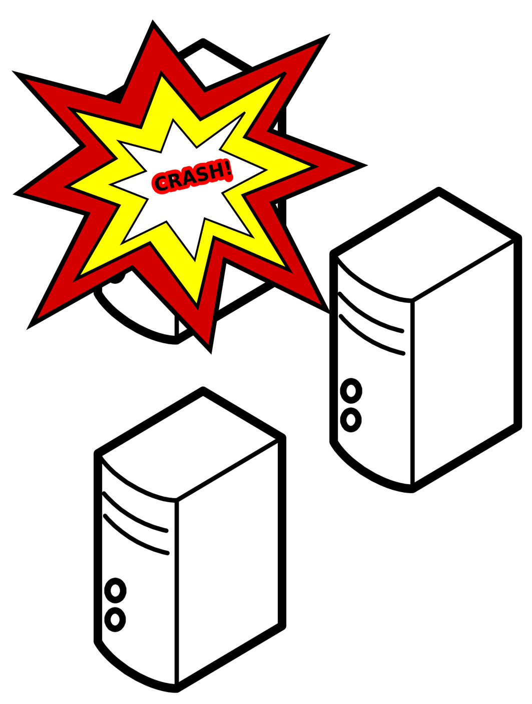
PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms

Cezara Drăgoi
INRIA, ENS, CNRS, France

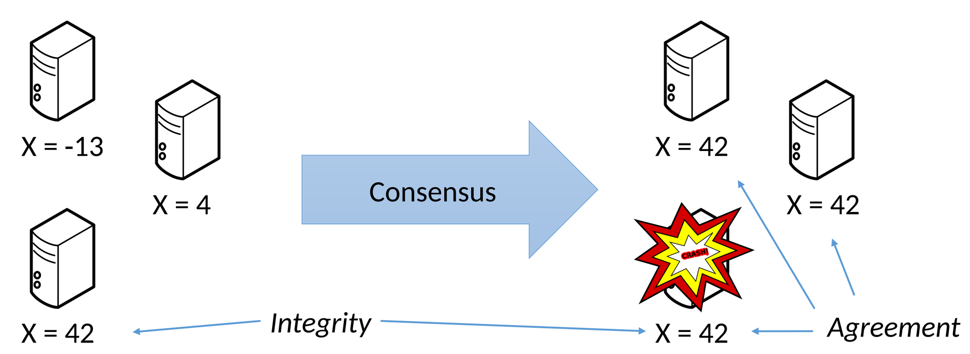
Thomas A. Henzinger
IST Austria, Austria

Damien Zufferey
MIT CSAIL, USA

Distributed systems use replications to withstand faults.



Fault-tolerant algorithms are used to maintain the global state consistent, despite process crashes and message delays. Consensus is a fundamental consistency problems. State-machine replication is built on top of consensus algorithms.



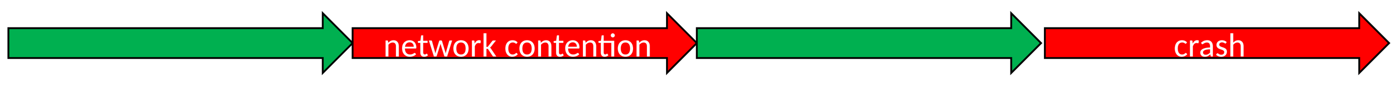
Irrevocability: Every correct process **decides at most one value.**
Termination: Every correct process **eventually decides.**

Network Model and Assumptions

[FLP 85]
asynchrony ∧ faults ⇒ consensus is not solvable

Some notion of time is needed to distinguish between processes crashing and message delays.

If the network is partially synchronous, i.e., it alternates between good (synchronous) and bad (asynchronous) periods, then consensus is solvable [Dwork et al. 88].



Programming Model

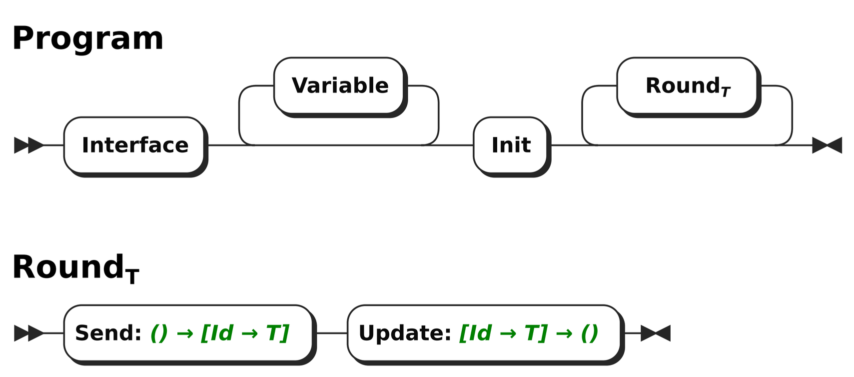
PSync has a lockstep semantics that gives the **illusion of synchrony**.

PSync programs are structured in **communication-closed rounds**.

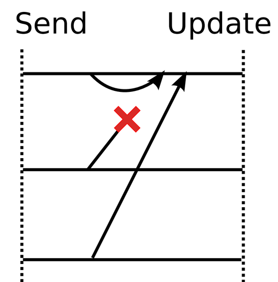
Faults are modeled by an **adversary who drops messages**.

The programming abstraction is based on the Heard-Of model [Charron-Bost & Schiper 09].

Abstract Syntax

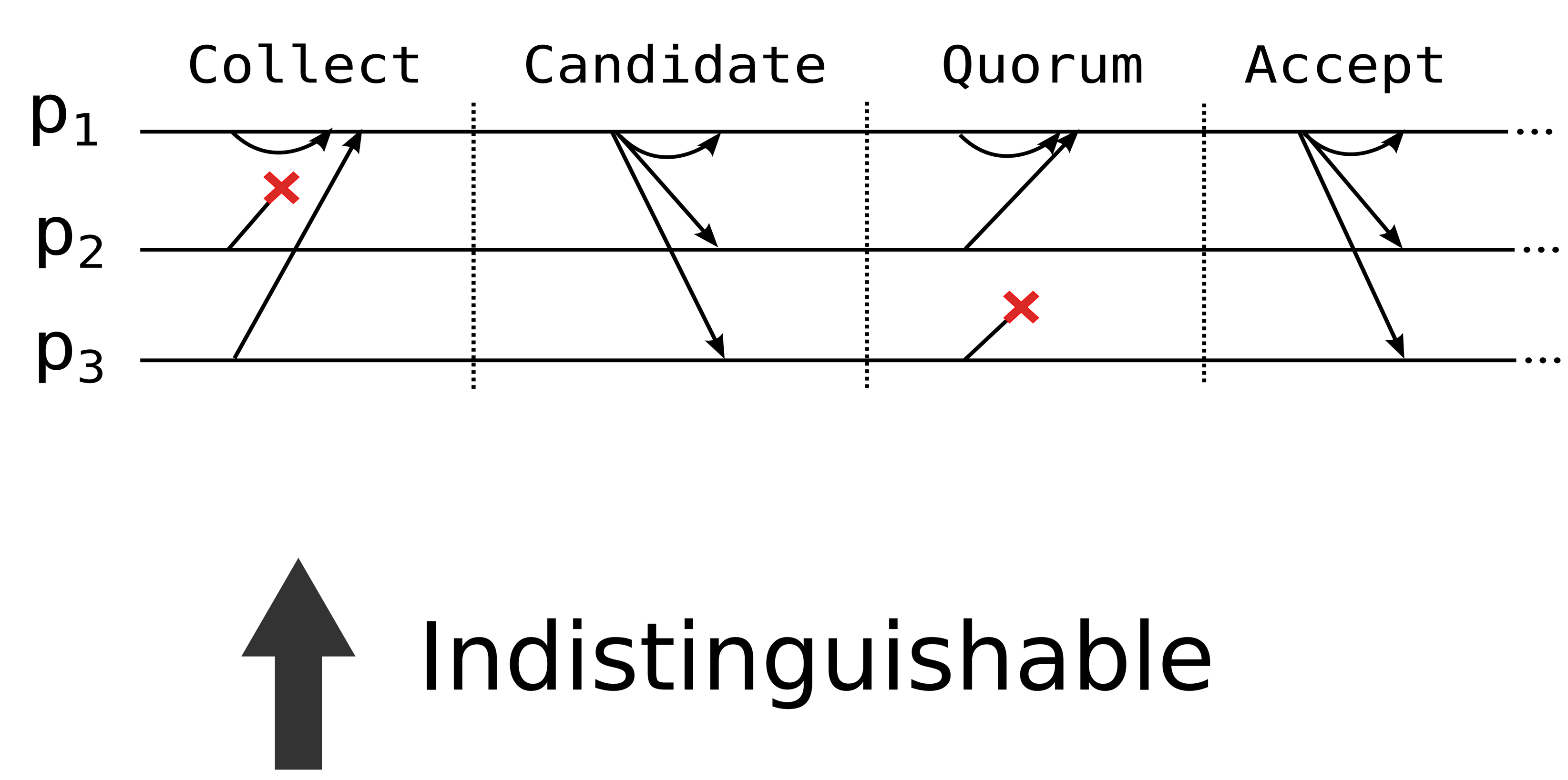


Semantics of a Round



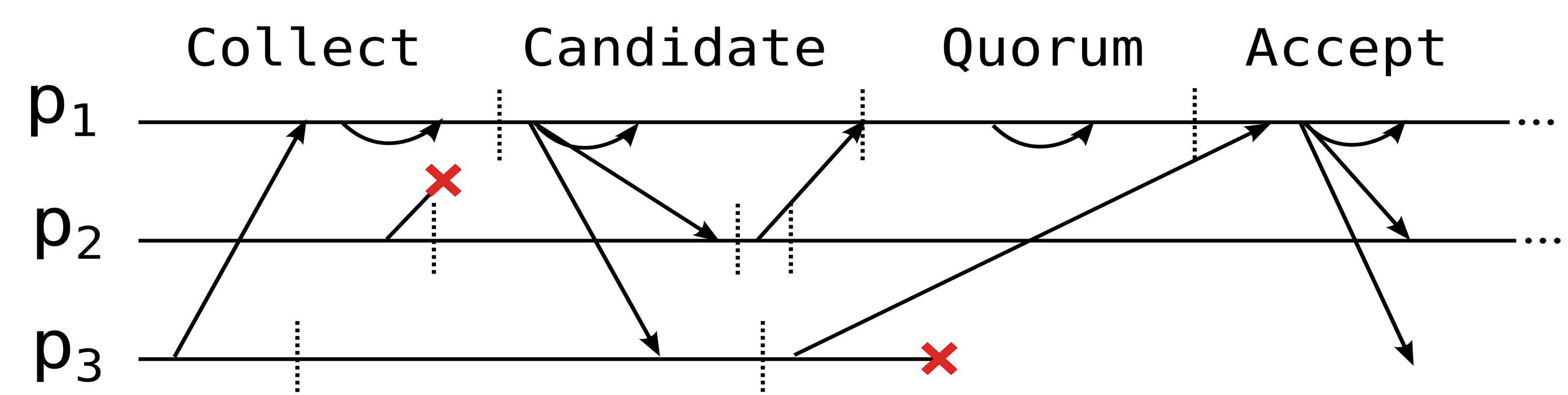
The adversay decides which messages are deliverd.

Lockstep Semantics



Indistinguishable

Asynchrounous Execution



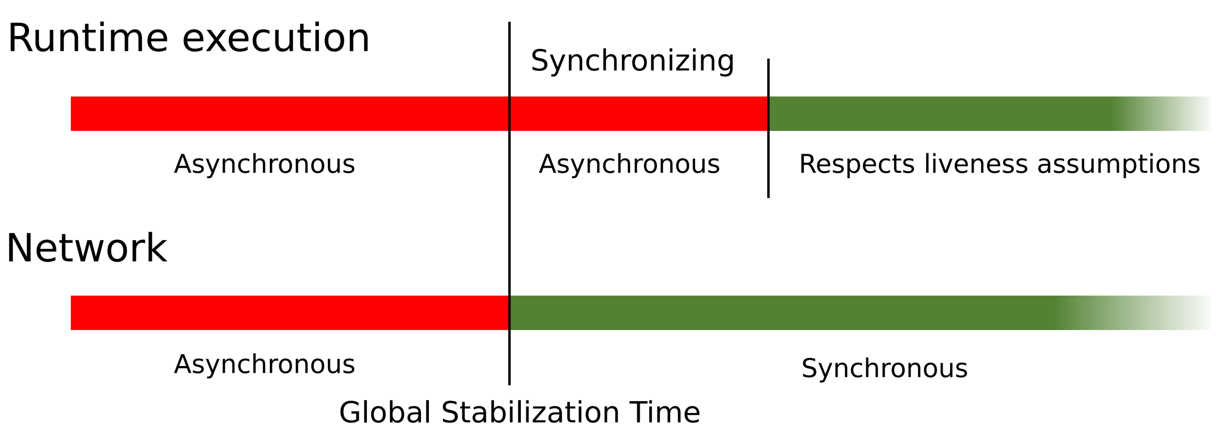
LastVoting Algorithm

Paxos-like algorithm in PSync

```
interface
    init(v: Int); out(v: Int)

variable
    x: Int; ts: Int; vote: Int
    ready: Boolean; commit: Boolean
    decided: Boolean; decision: Int

//auxiliary function: rotating coordinator
def coord(phi: Int): ProcessID =
    new ProcessID((phi/phase.length) % n)

//initialization
def init(v: Int) =
    x := v
    ts := -1
    ready := false
    commit := false
    decided := false

 Round /* Collect */ {
   def send(): Map[ProcessID, (Int,Int)] =
       return MapOf(coord(r) → (x, ts))
   def update(mbox: Map[ProcessID, (Int,Int)]) =
       if (id = coord(r) ∧ mbox.size > n/2)
           vote := mbox.valWithMaxTS
           commit := true },

 Round /* Candidate */ {
   def send(): Map[ProcessID, Int] =
       if (id = coord(r) ∧ commit) return
            broadcast(vote)
       else return ∅
   def update(mbox: Map[ProcessID, Int]) =
       if (mbox contains coord(r))
           x := mbox(coord(r))
           ts := r/4 },

 Round /* Quorum */ {
   def send(): Map[ProcessID, Int] =
       if ( ts = r/4 ) return MapOf(coord(r) → x)
       else return ∅
   def update(mbox: Map[ProcessID, Int]) =
       if (id = coord(r) ∧ mbox.size > n/2)
           ready := true },

 Round /* Accept */ {
   def send(): Map[ProcessID, Int] =
       if (id = coord(r) ∧ ready) return broadcast(vote)
       else return ∅
   def update(mbox: Map[ProcessID, Int]) =
       if (mbox contains coord(r) ∧ ¬decided)
           decision := mbox(coord(r))
           out(decision)
           decided := true
       ready := false
       commit := false }
```

Asynchronous Runtime

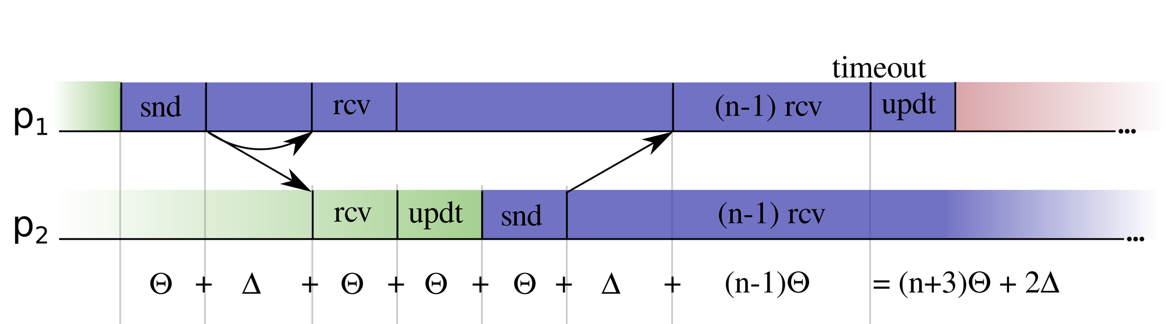
During long enough good periods, the runtime preserves the liveness assumptions.



The Runtime is based on *timeouts*.

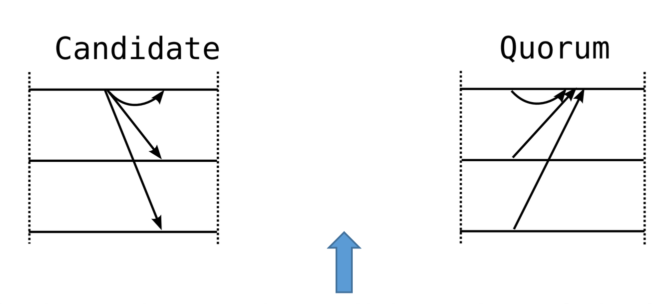
During good periods we assume:
  Θ is the minimal interval in which any process takes a step;
  Δ is the maximal transmission delay between any two processes.

We can compute the minimal timeout needed to guarantee progress:



Benefits for Verification

Round structure	⇒	reasoning about rounds in isolation
Communication-closed rounds	⇒	no message in flight between rounds
Lockstep semantics	⇒	no interleaving



Simple invariants describing the global system at the boundaries between rounds.

Invariant to show agreement in LastVoting

$$\forall i.\, \neg decided(i) \wedge \neg ready(i)$$

$$\begin{aligned}
\exists\, v, t, A.\ A = \{\, i.\ ts(i) > t \,\} \wedge |A| > n/2 \\
\wedge \quad & \forall i.\, i \in A \Rightarrow x(i) = v \\
\wedge \quad & \forall i.\, decided(i) \Rightarrow x(i) = v \\
\wedge \quad & \forall i.\, commit(i) \vee ready(i) \Rightarrow vote(i) = v \\
\wedge \quad & t \leq \Phi \\
\wedge \quad & \forall i.\ ts(i) = \Phi \Rightarrow commit(coord(i)) = v
\end{aligned}$$

Implementation

https://github.com/dzufferey/psync
Embedding in Scala, Apache 2.0 License.

Implementation of multiple fault-tolerant distributed algorithms in PSync.

Verification condition generator using user provided invariants.

Paxos case study:
  Conciseness against other DSLs for distributed algorithms;
  Efficiency against low-level implementations.

Thomas A. Henzinger 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). Damien Zufferey was supported by DARPA (Grants FA8650-11-C-7192 and FA8650-15-C-7564) and NSF (Grant CCF-1138967).

[Charron-Bost & Schiper 09]
B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. Distributed Computing, 2009.

[FLP 85]
M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM, Apr. 1985.

[Dwork et al. 88]
C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. J. ACM, 1988.