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

Authors: Cezara Dragoi, Thomas A. Henzinger, Damien Zufferey

Presented by: Abhishek A. Singh

PSYNC

1

Domain Specific Language

2

Based on Heard-Of Model

- Which views asynchronous faulty systems as synchronous systems with an adverserial component

3

Implemented as an embedding in Scala

4

Created for modelling, programming& verification of distributed fault-tolerant algorithms

Heard-of Model

- Algorithms structured in communication closed rounds.
- Each round consists of two consecutive operations: Send and Update
- Two components in the model:
  - Set of Processes
  - Adversarial environment: which determines whether messages are received or not
- Each process has a Heard-of set: HO
- Each round is communication closed: all messages sent in a round are either received or dropped

The Heard-of set (HO)

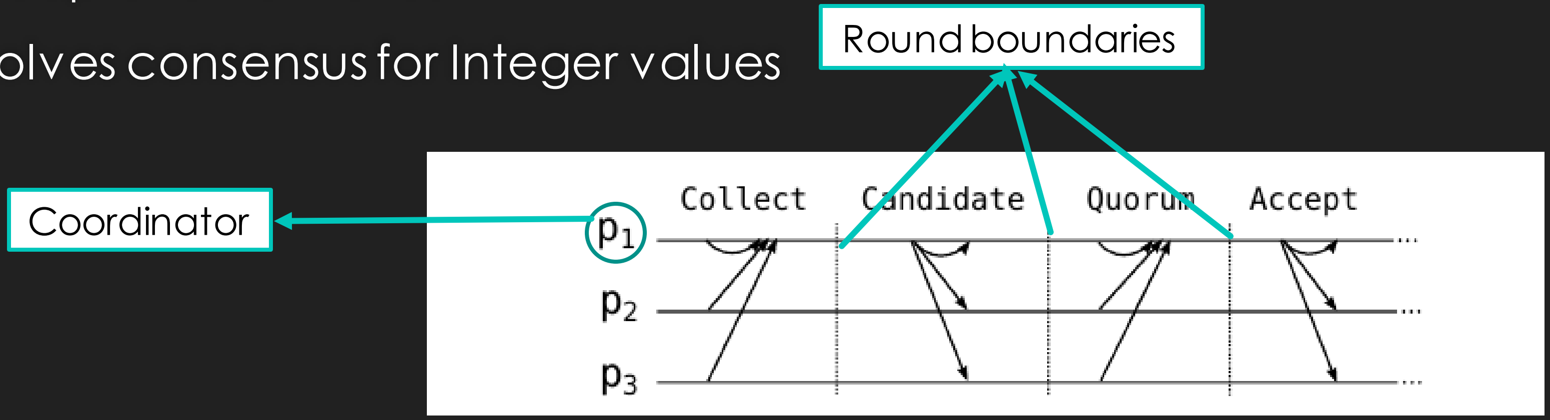
- Abstraction for the asynchronous and faulty behavior of the network.
- HO Is a set of processes.
- In a round:
  - Process ***p*** receives a message from ***q*** if ***q*** sent a message to process ***p*** and ***q*** $\in$ HO(***p***)
- HO+Rounds => Abstract notion of time + control structure for programmers

Runtime model

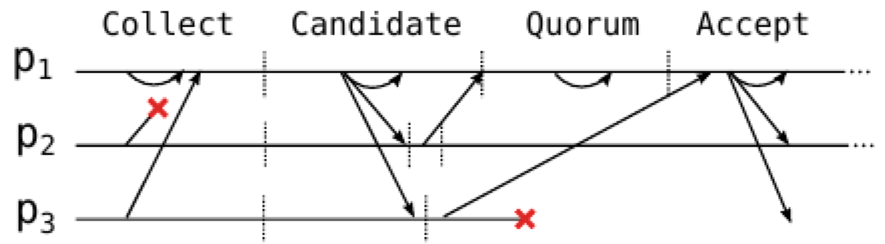
- Based on timeouts
- Send, Wait, Update, Move to next round, Repeat

Example: LastVoting

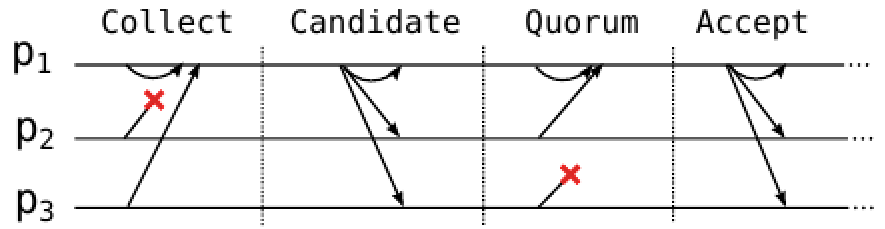
- Adaptation of Paxos
- Solves consensus for Integer values



LastVoting: Execution



(a) An asynchronous, faulty execution of the *LastVoting*



(b) Corresponding indistinguishable lockstep execution

LastVoting: 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
```

```
val phase = Array[Round]( //the rounds
  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 })
```

PSync Syntax

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{interface}\ \mathit{variable}^*\ \mathit{init}\ \mathit{phase} \\
\mathit{interface} & ::= & \texttt{init}\colon \mathit{type} \rightarrow ()\quad (\mathit{name}\colon \mathit{type} \rightarrow ())^* \\
\mathit{variable} & ::= & \mathit{name}\colon \mathit{type} \\
\mathit{init} & ::= & \texttt{init}\colon \mathit{type} \rightarrow () \\
\mathit{phase} & ::= & \mathit{round}^+ \\
\mathit{round}_{\mathsf{T}} & ::= & \texttt{send}\colon () \rightarrow [P \mapsto \mathsf{T}]\quad \texttt{update}\colon [P \mapsto \mathsf{T}] \rightarrow ()
\end{array}
$$

Lockstep execution

Definition 5 (Lockstep execution)**.** *Given a* PSYNC *program* $\mathcal{P}$ *and a non-empty set of processes* P, *a lockstep execution of* $\mathcal{P}$ *is the sequence* $*A_0 s_1 A_1 s_2 \ldots$ *such that*

- $*A_0 s_1$ *is the result of the* INIT *rule;*
- $\forall i.\ s_i A_i s_{i+1}$ *satisfy the* SEND *or the* UPDATE *rule;*
- *the environment assumptions on* HO*-sets are satisfied.*

The set of lockstep executions of $\mathcal{P}$ *is denoted by* $[\![\mathcal{P}]\!]_{ls}$.

Indistinguishability

- Defined in terms of a transition system.
- Transition system is intended to reflect an instance of execution

Definition 1 (Indistinguishability). *Given two executions π and π' of a transition system TS, a process p cannot distinguish locally between π and π', denoted $\pi \simeq_p \pi'$, iff the projection of both executions on p agree up to finite stuttering, i.e., $\pi|_p \equiv \pi'|_p$.*

Two executions π and π' are indistinguishable, *denoted $\pi \simeq \pi'$, iff no process can distinguish between them, i.e., $\forall p \in P.\ \pi \simeq_p \pi'$.*

Definition 2 (Indistinguishable systems). *A system TS_1 is indistinguishable from a system TS_2 denoted $TS_1 \trianglerighteq TS_2$ 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 $\pi \simeq_{W,L} \pi'$ where $W = V_1 \cap V_2$ and $L = A_1 \cap A_2$.*

Distributed clients

Definition 3 (Distributed client). [1] *Let* $TS_i = (\{p_i\}, V_i, A_i, s_0^i, T_i)$ *be the transition system associated with a client process, with* $A_i \cap A_j = \emptyset$ *for all* $1 \leq i \neq j \leq n$. *Formally, the transitions system associated with the client is* $C_{TS} = (P, V, A, s_0, T)$, *where* $P = \{p_1, p_2, \ldots p_n\}$, $V = \biguplus_i V_i$, $A = \bigcup_i A_i$, $s_0 = (s_0^1, \ldots, s_0^n)$, *and* $T \subseteq \Sigma \times A \times \Sigma$, *with* $\Sigma = [P \to V \to \mathcal{D}]$, *such that* $\Sigma(p_i) \in [V_i \to \mathcal{D}]$, *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.*

- All distributed clients as commutative by definition

Observational refinement

Definition 4 (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,*

$$\mathsf{Runs}(C(TS_1)) \subseteq \mathsf{Runs}(C(TS_2)).$$

- TS1 observationally refines TS2 if every run of a client that uses TS1 is also a run of the same client using TS2.

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

- Indistinguishability is equivalent with sequential consistency

Comments or Questions