Implementing Communication-Closed Rounds: Toward an Efficient and General Solution

Damien Zufferey (MPI-SWS)
FRIDA, 2020.09.04
joint work with Cezara Dragoi (INRIA)
and Josef Widder (Informal Systems)

Back to 1st FRIDA in 2014

Small trip down memory lane… Back in 2014 we were already looking at this question:

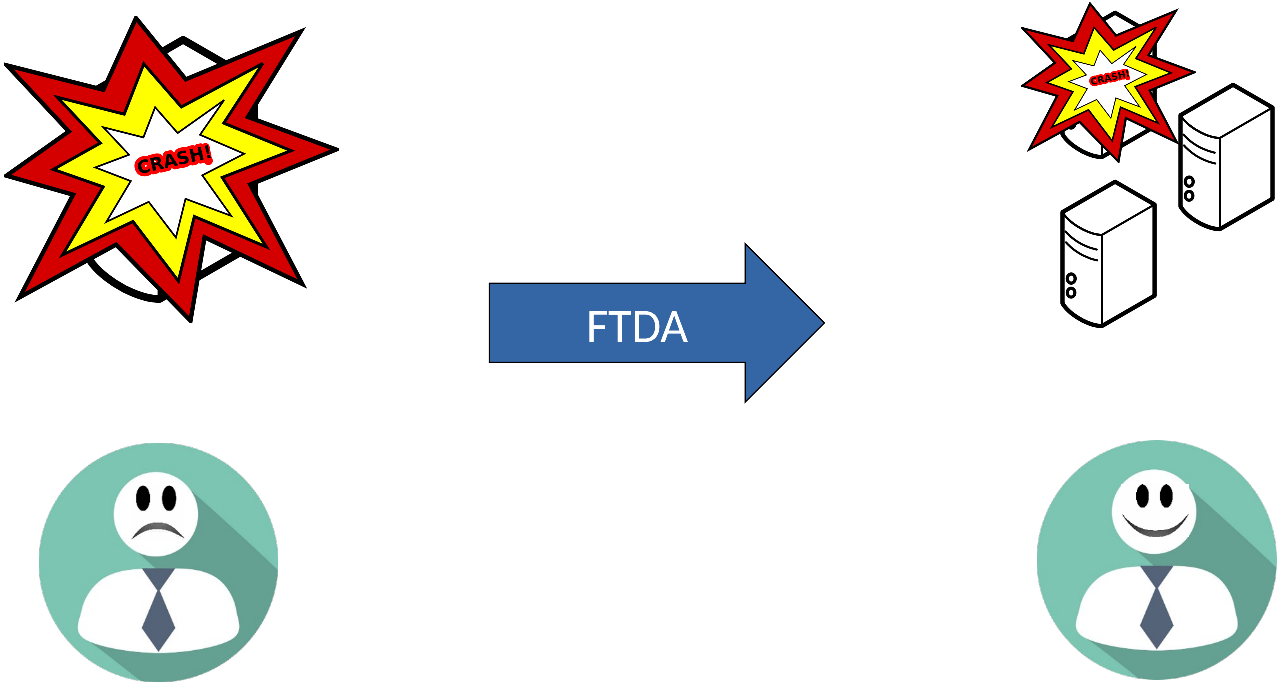
https://github.com/dzufferey/presentations/blob/master/2014_07_24_FRIDA/Round%20model%20for%20DA.pdf

Progress has not always been fast but we are getting there :)

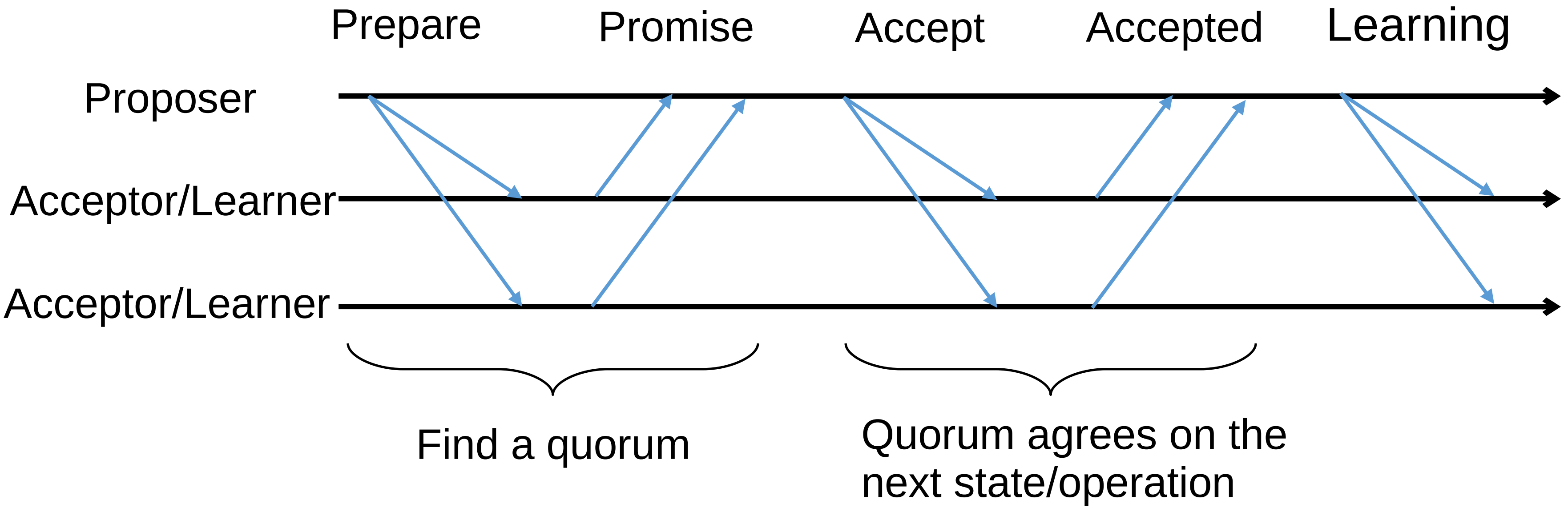
Outline

- Faut-tolerant distributed algorithm (FTDA) and their implementation
- Round models: benefits, shortcoming, and implementation
- Toward a configurable round model
- Results

Why FTDA?



The Paxos Algorithm

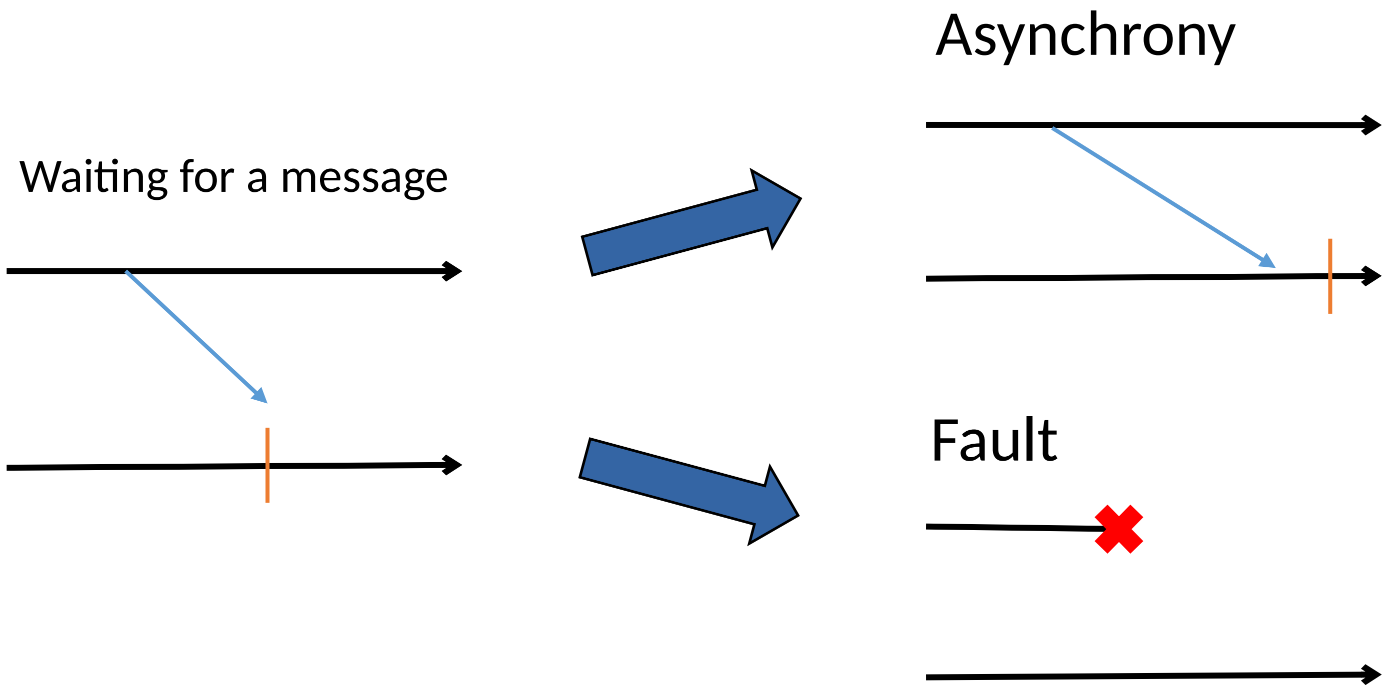


Implementing Paxos: from ~50 lines of pseudo code to >500 LoC. What goes wrong?

Implementation Challenges

- Detecting failure (and the impossibility to get it right)
- Messages (side-effects) are untyped, have no scope, etc.
- Control-flow inversion (losing the program structure)

When Processes Fail?



Wait longer can help but how to avoid blocking?

Asynchronous models are not good at dealing with faults.

[FLP 85] Solving consensus requires time.

Communication is a Side Effect

```
Type object;
Byte[] buffer =
      serialize(obj);
send(channel, buffer);
```

011001010101…

```
buffer = recv(channel);
Type object1 =
    deserialize1(buffer);
…
buffer = recv(channel);
OtherType object2 =
    deserialize2(buffer);
```

Up to the programmer to:
- interpret the bytes moving over the network,
- know which receive corresponds to which send.

Control-flow Inversion

Protocol structure replaced by dispatch:

Protocol:
(1)Msg A
(2)Msg B

```
var state = 1

while (true) {
    on receive {
        case Msg A =>
            if (state == 1) …
            else if (state == 2) …
        case Msg B =>
            if (state == 1) …
            else if (state == 2) …
    }
}
```

normal case

message duplicated

message dropped

normal case

Round Model: (Pseudo)code

Algorithm 9 The *OneThirdRule* algorithm

Initialization:
  $x_p := v_p$ $\{v_p$ is the initial value of $p\}$

Round r**:**
  S_p^r :
    send $\langle x_p \rangle$ to all processes

  T_p^r :
    if $|HO(p,r)| > 2n/3$ **then**
      $x_p :=$ the smallest most often received value
      if more than $2n/3$ values received are equal to $\overline{x}$ **then**
        DECIDE($\overline{x}$)

From Charron-Bost and Schiper (2009)

```
class OtrProcess extends Process {

  var x = ???

  val rounds = phase(
    new Round[Int]{

      def send(): Map[ProcessID,Int] = broadcast(x)

      def update(mailbox: Map[ProcessID,Int]) = {
        if (mailbox.size > 2*n/3) {
          x = minMostOftenReceived(mailbox)
          if (mailbox.filter( _._2 == x ).size > 2*n/3) {
            decide(x)
} ) } } } }
```

PSync code (POPL 2016), slightly abbreviated

Complete code at: https://github.com/dzufferey/psync/blob/master/src/test/scala/example/Otr.scala

Communication-Closed Rounds

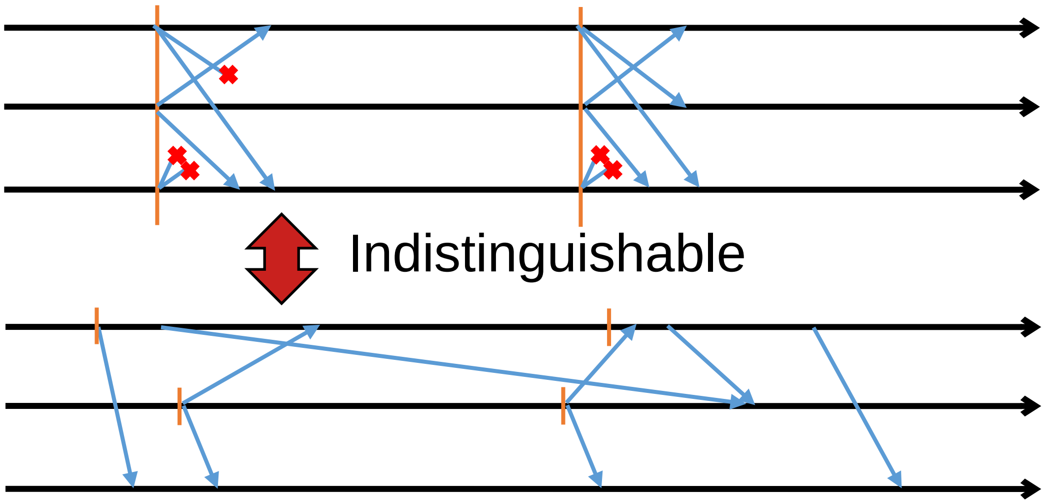
- Rounds are syntactic units that
  - Give a scope to messages (connect send receive)
  - Typed (serialization)
  - Failure detection provided by a dedicated runtime
- The implementation difficulty is still there but hidden within the runtime that provides the round abstraction.
- CC rounds also helps verification (not covered in this talk)

Round vs Real System

Idea: model faults/asynchrony as an adversarial environment [Gafni 98]
Project all the “faults” on the messages such that local views are preserved.

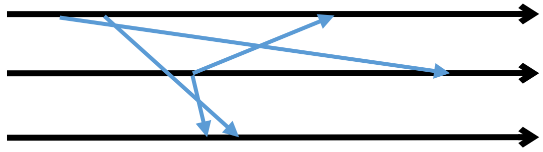
Lockstep semantics:

Asynchronous execution:

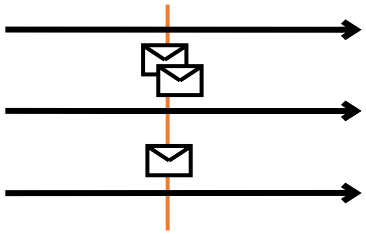


Benefits for the Verification

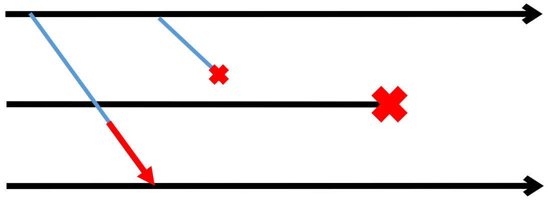
- Asynchrony (interleaving, delays)



- Channels

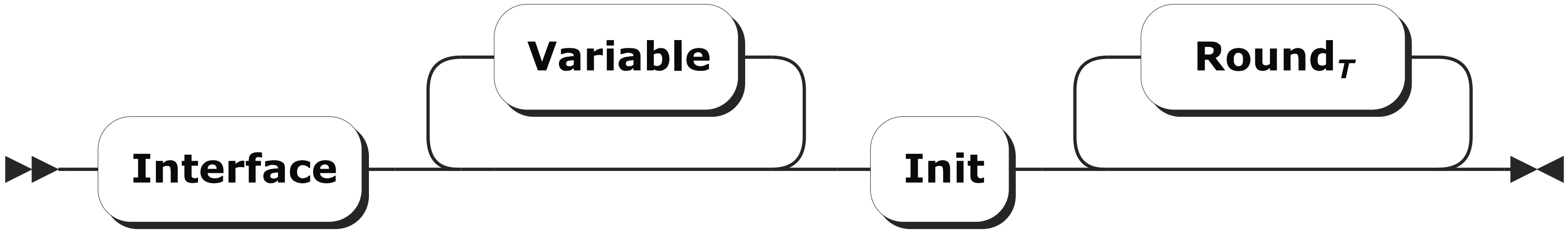


- Faults

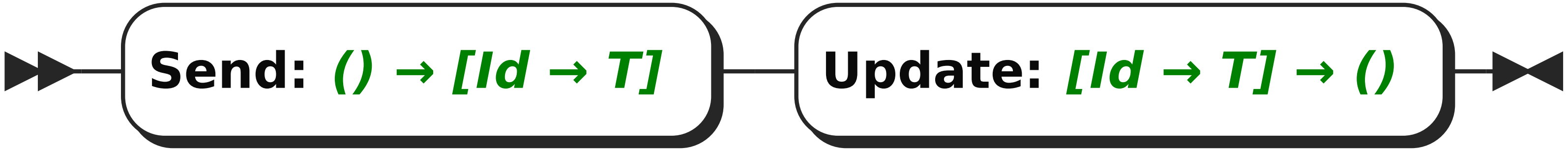


PSync Program Structure

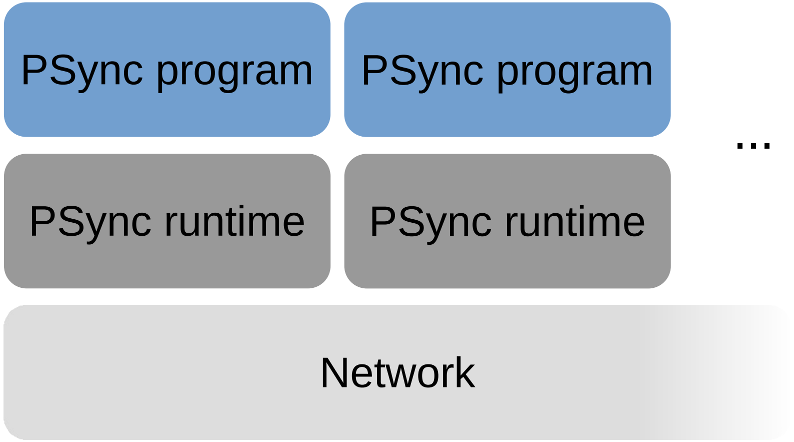
Program

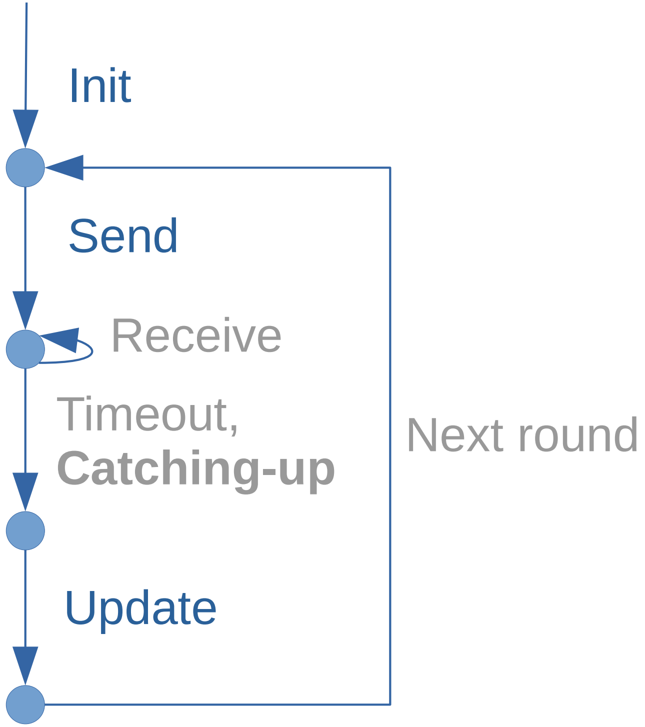


Round



PSync Runtime





The runtime deals with the network and hides events.

Liveness Guarantees

- PSync runtime only work for **partial synchrony** (Dwork et al. 1988).
  - During **bad period**, the processes can get arbitrarily desynchronized.
  - During **good period**, the processes work in lockstep.
- To resynchronize, slow processes need to catch-up, i.e., progress to the next round as soon as they receive a message from an higher round.

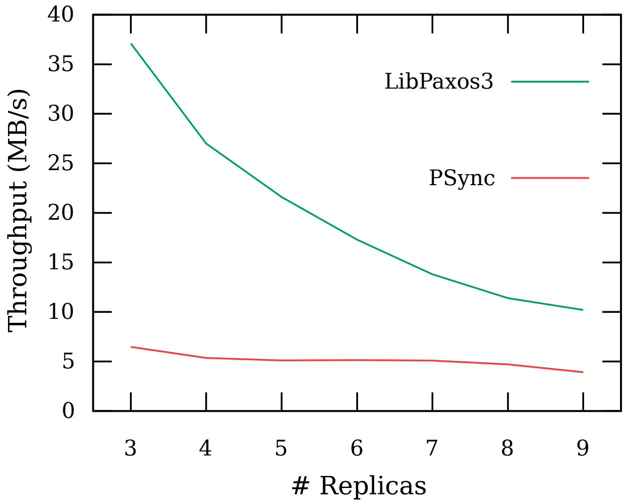
Programming with Rounds

For round to work as programming abstraction, we need:

	PSync
– Generality:	
• Algorithm	✓
• Fault model	✗
– Reasonable overhead:	
• Algorithm	✗
• Fault model	✗
• Deployment	✓ ✗

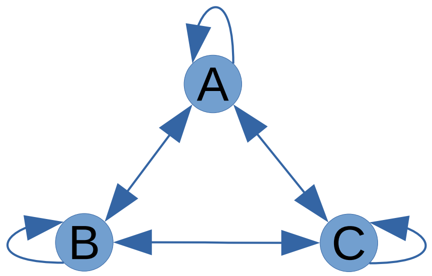
One of PSync's Limitations

- The rate of progress is limited by the timeout!
  - Small TO: faster but less resilient to jitter
  - Large TO: slower but more resilient to jitter
- Many algorithms only needs timeout to detect faults but could progress as soon as all the messages for a round are received.



All-to-All needs TO

Communication pattern

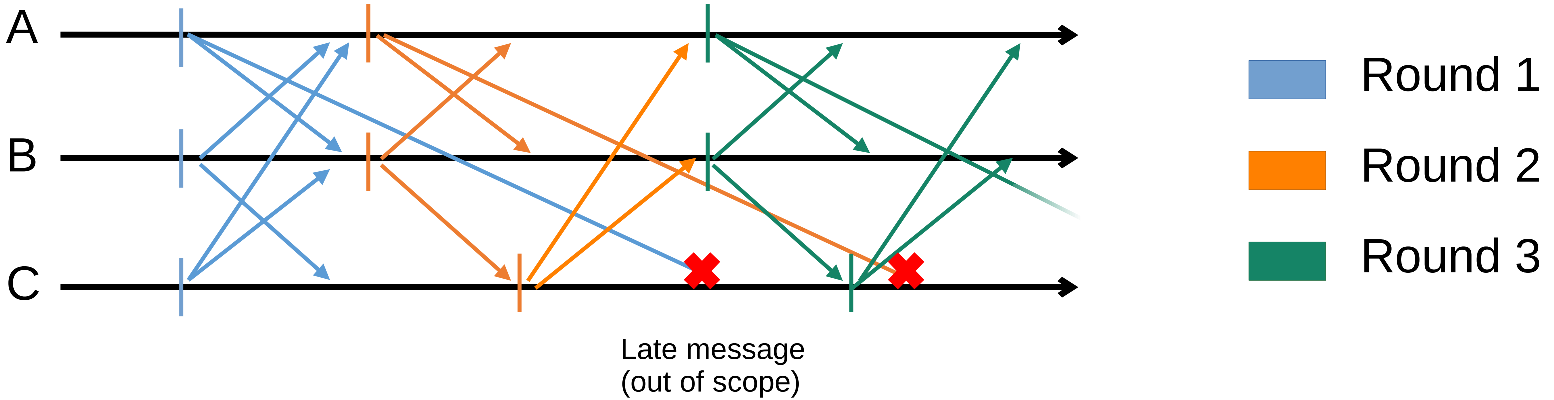


Message delays

Snd \ Rcv	A	B	C
A	0	1	**3**
B	1	0	1
C	1	1	0

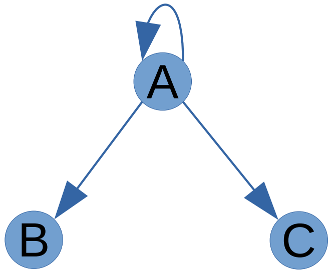
Assumptions

- All messages needed to progress.
- All processes start at “t = 0”.

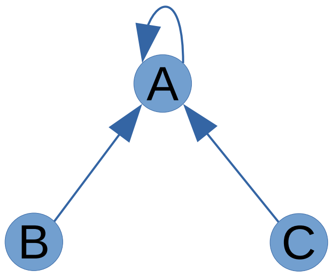


All-to-One/One-to-All w/o TO

Communication topology



and

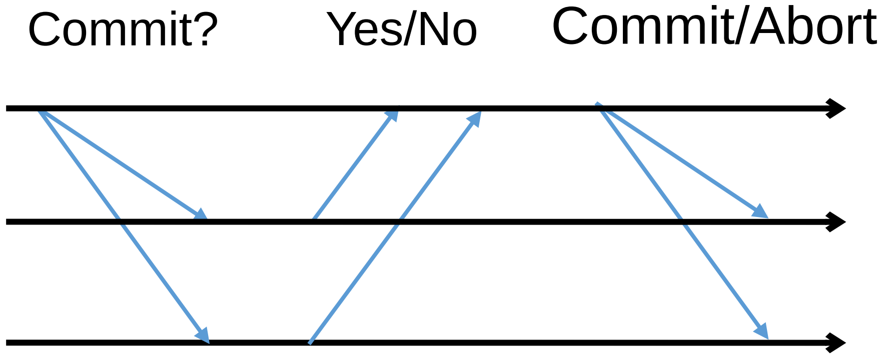


- Communication pattern for leader-based algorithms.
- The leader acts as “synchronization bottleneck.”
- Processes can proceed as soon as they have received the messages without compromising global progress.

Progress and Message Values

Round models process messages in a single batch for the whole round. For some algorithm, specific messages can trigger faster progress.

Two Phase Commit protocol (2PC)

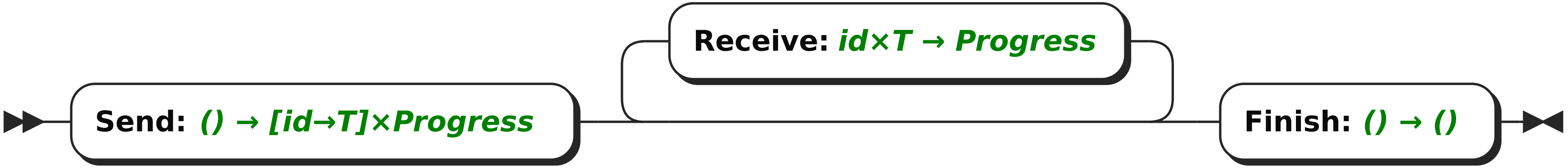


A single “No” in the 2nd round leads to “Abort”.

Giving Control to the Programmer

- Rather than case-splitting on the algorithm, let the programmer decide.
- The programmer knows *what the algorithm needs.*
- The programmer knows *the deployment scenario.*

New Round



Progress hints to tell the runtime what to do. Progress has two parts:

1) When to finish a round?

GoAhead	timeout ≤ 0
Timeout *t*	timeout $\in (0, \infty)$
WaitMessage	timeout $= \infty$

2) Automatic resynch. ?

- Allow catch-up (default)
- Block catch-up

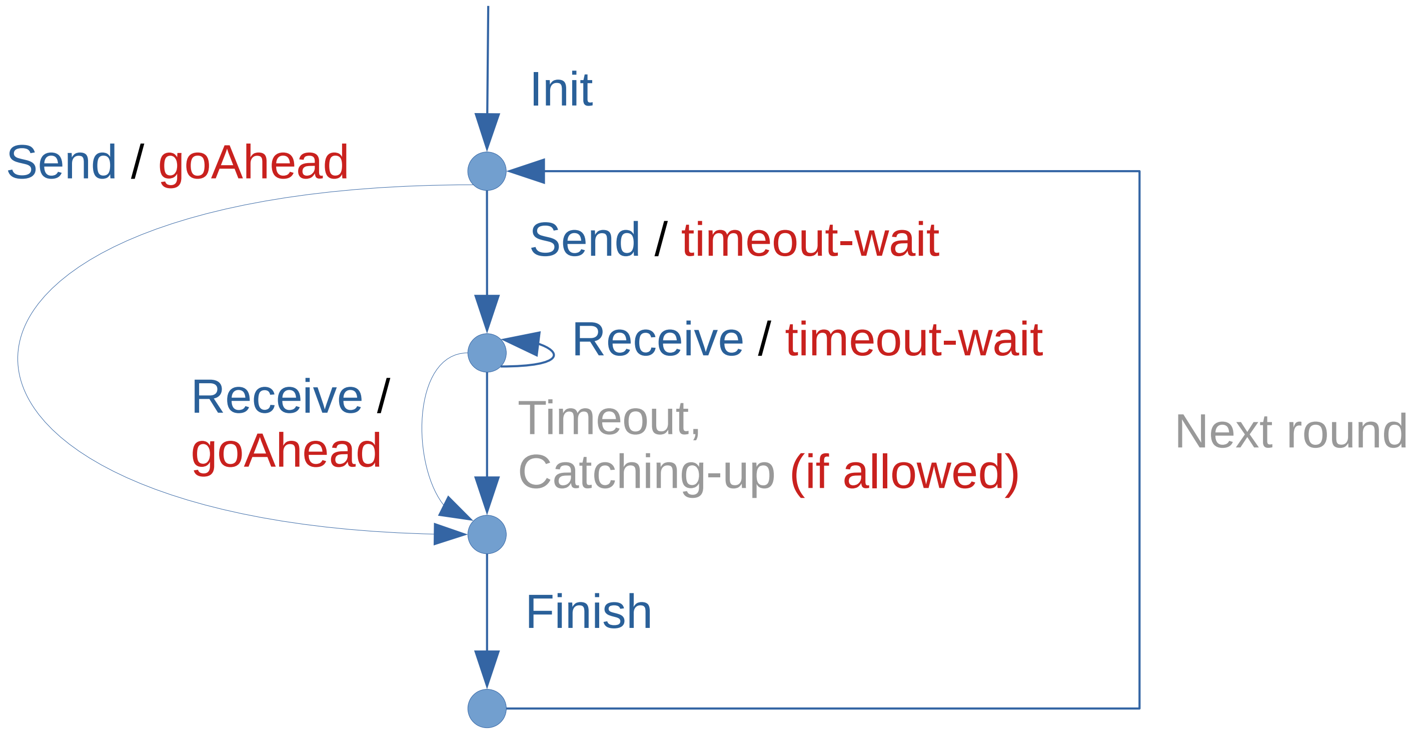
Old vs New: 2nd Round of 2PC

```
new Round[Boolean](timeout){

  def send(): Map[ProcessID,Boolean] = {
    Map( coord -> vote )
  }

  def update(mailbox: Map[ProcessID,Boolean]) = {
    if (id == coord) {
      commit = mailbox.size == n && mailbox.forall( _._2 )
    }
  }
}
```

The new version is more complex but it gives more control. When a “false” message is received the coordinator can progress.

```
new Round[Boolean]{

  var nMsg = 0
  var ok = true

  def send(): (Map[ProcessID,Boolean], Progress) = {
    val msg = Map( coord -> vote )
    val prog =  if (id != coord) Progress.goAhead
                else Progress.timeout(timeout)
    (msg, prog)
  }

  def receive(sender: ProcessID, payload: Boolean) = {
    nMsg += 1
    ok &= payload
    if (!ok || nMsg == n) Progress.goAhead
    else Progress.timeout(timeout)
  }

  def finishRound() = {
    if (id == coord) commit = ok && nMsg == n
  }
}
```

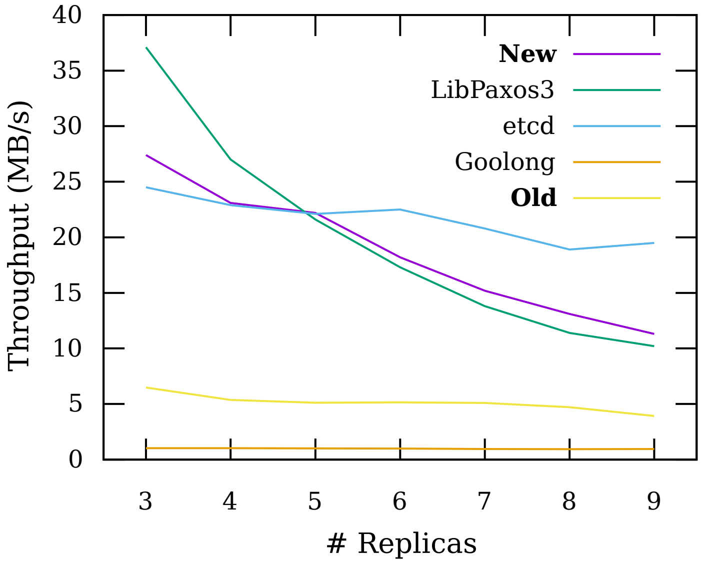
New Runtime



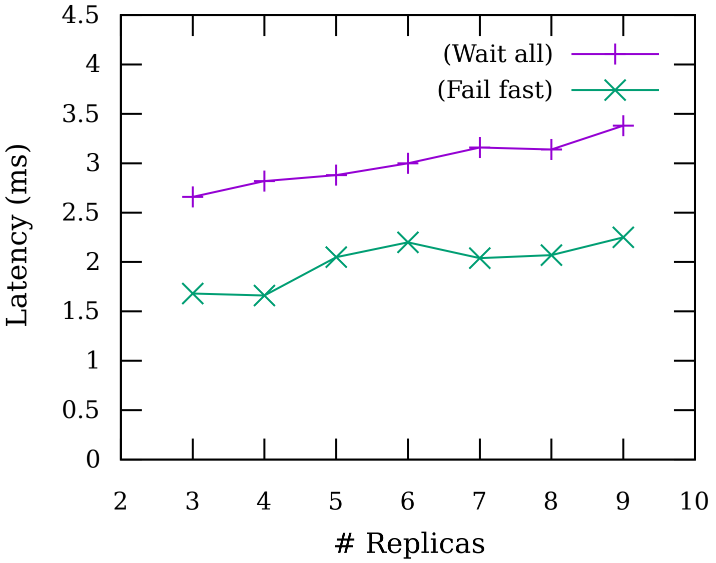
Init
Send / goAhead
Send / timeout-wait
Receive / timeout-wait
Receive / goAhead
Timeout, Catching-up (if allowed)
Next round
Finish

In Practice

Throughput for Paxos style consensus

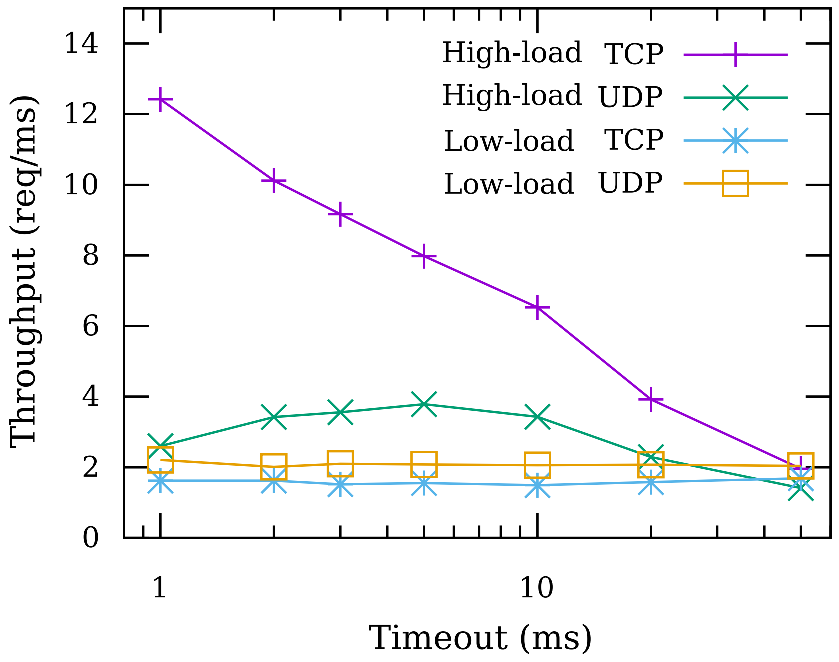


Latency for 2PC



Deployment is also Important

The algorithm is one part, tuning parameters like timeout, transport layer, etc., also has a large impact.



About Byzantine Faults

- Byzantine Faults not cover due to time…
- Progress abstraction can work with Byzantine faults:
  - Need to update the catch-up mechanism.
  - Add primitive to block until enough processes have reached a certain round.
- Stay tuned for the paper for the full explanation.

Conclusion

- Communication-closed rounds are a good abstraction for FTDA (simplify programming and verification)
- For more generality and performances, the programmer needs more control over the runtime.
  - Progress indication for timeout and resynchronization
- Implemented in https://github.com/dzufferey/psync