Swift Algorithms for Repeated Consensus *

Fatemeh Borran Martin Hutle Nuno Santos André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
{*firstname*}.{*lastname*}@*epfl.ch*

Abstract

We introduce the notion of a swift algorithm. *Informally, an algorithm that solves the repeated consensus is swift if, in a partial synchronous run of this algorithm, eventually no timeout expires, i.e., the algorithm execution proceeds with the actual speed of the system. This definition differs from other efficiency criteria for partial synchronous systems.*

Furthermore, we show that the notion of swiftness explains the reason why failure detector based algorithms are typically more efficient than round-based algorithms, since the former are naturally swift while the later are naturally non-swift. We show that this is not an inherent difference between the models, and provide a round structure implementation that is swift, therefore performing similarly to failure detector algorithms while maintaining the advantages of the round model.

1 Introduction

Timeouts are often required to solve problems in distributed computing. Due to the FLP impossibility result [7], there is a need of some minimal synchrony assumptions for solving the consensus problem, and timeouts are the dominant mechanism for algorithms to make use of synchrony assumptions.

Timeouts are often chosen conservatively, so that the algorithm is correct for a large number of real-life scenarios. However, timeouts should be used only to cope with faults, and not slow down the execution time in good cases. As an example, when implementing communication-closed synchronous rounds in a synchronous message passing system, after a process sends its messages for a certain round it usually waits for a timeout, before it ends the round and sends its messages for the next round. However, in many runs of the algorithm, it might have received all messages from other alive processes already long before that. It would be favorable to start the next round immediately after all messages from correct processes are received. This is, for example, the case with an algorithm that uses a $\Diamond\mathcal{P}$ failure detector (FD). Here, a process waits for a message from some process p until p is in the FD output. If p has crashed, this involves waiting for a timeout, but only once: later rounds profit from the fact that the failure detector "remembers" information about faults. We formally capture such a behavior by the definition of *swift*, which we define in the context of repeated consensus [5]. The main intuition behind our definition is that swift algorithms make progress at the speed of the system, and therefore, are more "efficient" than non-swift algorithms. A swift algorithm for a repeated problem is thus one in which eventually all instances of the problem are "efficient".

In more detail, for the definition of swift we look at partial synchronous runs, *i.e.*, runs where a bound Δ on the transmission delay eventually holds forever.[1] For the good period of such a run, that is the partial run R in which bound Δ holds, we can define the actual transmission delay $\delta(R)$ as the maximum of all transmission delays in R. Such an actual transmission delay can be much smaller than the maximum transmission delay Δ. If in this case, the execution time for each instance of the repeated consensus eventually depends only on $\delta(R)$ (in contrast to Δ), the algorithm is *swift*.

While intuitively swift algorithms progress at the speed of messages in good periods, and non swift algorithms progress only at the speed of expiration of timeouts, we refrained from calling these two classes of algorithms *message-driven* and *timeout driven*. This is because the term *message-driven* is used in [10, 1] with a different meaning, namely to refer to the way events are generated at a process. If processes are allowed to measure time (*e.g.*, with clocks or step counting), then it is possible to construct message-driven algorithms (according to this definition) that are not swift. On the other hand, if processes use an adaptive timeout, then the algorithm can be swift despite timeout expiration. Thus these terms are not suitable to precisely characterize this class of algorithms.

*Research funded by the Hasler Foundation under grant number 2070.

[1] Note that such a run exists also, *e.g.*, in an asynchronous system, and all runs of a synchronous systems are of course also partial synchronous. The definition is thus not limited to partial synchronous systems.

Other notions of efficiency for distributed algorithms have been considered. The term *fast* has been used to refer to (consensus) algorithms that solve consensus with less communication steps in favorable cases [12]. A favorable case corresponds usually to an execution without faults that is synchronous from the beginning. On the contrary, the definition of swift is related to the execution *time* of an algorithm in the context of *repeated* consensus. Further, the definition of swift considers also runs with faults. The notion of fast is orthogonal to the notion of swift: it is possible to design both fast algorithms that are swift and fast algorithms that are not swift.

The paper makes the following two contributions. The first contribution is the definition of swift algorithms that we just discussed. The second contribution is a new implementation of a communication-closed rounds in a partial synchronous system with crash faults. This new implementation leads to swift round-based consensus algorithms, while previous round implementations, including those described in [6, 8], are not swift. This result is highly relevant in the context of comparing advantages and drawbacks of the failure detector approach [3] vs. the round-based approach [6, 4] for solving agreement problems. Indeed, failure detector based algorithms, despite the usage of timeouts in the implementation of the failure detector algorithm, are naturally swift. On the other hand, round implementations in a partial synchronous model have some advantages over FD based implementations [9]. Our new solution thus combines the advantages of both approaches.

The rest of the paper is structured as follows. In the next section, we specify our model and give a formal definition of *swift*. Then, in Section 3 we show a simple round-based consensus algorithm that is not swift, and in Section 4 we show that the same consensus algorithm expressed using a failure detector is swift. In Section 5 we present our main contribution: we show a new implementation of rounds that is swift. Section 6 validates the theoretical analysis with experimental results comparing the swift and non-swift implementations. Section 7 concludes the paper.

2 Definitions and model

We consider a system of n processes connected by a message-passing network. Among these n processes, at most f may crash. For repeated consensus, we attach an in-queue and an out-queue to each process. Processes execute an algorithm by making steps, where a step can be either a send step $\langle p, \text{SEND}, m\rangle$, in which a process sends a message to another process, a receive step $\langle p, \text{RECEIVE}, S\rangle$, in which a (possibly empty) set S of messages is received, an input step $\langle p, \text{IN}, I\rangle$ in which a value is read from p's in-queue, or an output step $\langle p, \text{OUT}, O\rangle$, in which a value is output to p's out-queue. We denote with In_p (resp. Out_p) the in-queue (resp. out-queue) of process p. In each step a process also performs a state transition.

We assume an abstract global discrete time. Without loss of generality, at each time t at least one process makes a step. A single process can make at most one step at any time. Processes measure time by counting their own steps.

Channels satisfy validity and integrity.[2] Channels are reliable if additionally the following property holds:

Reliability: If message m is sent from p to q and q performs an infinite number of receive steps, then eventually m is received by q.

We consider partial synchronous runs, defined by a bound Φ on the process relative speeds and a bound Δ on the transmission delay of messages [6]. For a run R, we say that the process speed bound Φ holds in R if in any partial run of R that contains Φ steps, every non-crashed process makes at least one step. Further, we say that the transmission delay Δ holds in R after some time t_0 if (i) any message sent by p to q at time $t \geq t_0$ is received the latest in the first receive step after $t + \Delta$; and (ii) every message that sent before t_0 is received the latest in the first receive step after $t_0 + \Delta$.

Definition 1 (Partial synchrony)**.** *A run R is* Δ-partial synchronous *if there is a time GST (Global Stabilization Time) such that after GST the transmission delay bound Δ holds, the process speed bound Φ holds, and no process crashes after GST.*

We call the time interval $[GST, \infty]$ the *good period of* R. We say a *system* is *Δ-partial synchronous* if every run R of the system fulfills Definition 1. To simplify the presentation, we assume $\Phi = 1$.

Definition 2 (Actual parameters)**.** *Let R' be a partial run. Then $\delta(R')$ denotes the maximum transmission delay of the partial run R', i.e., the smallest value $\overline{\delta}$ such that the transmission delay is bounded by $\overline{\delta}$ in the partial run R'.*

If R' is the good period of a Δ-partial synchronous system, then $\delta(R') \leq \Delta$. When R' is clear from the context, we simply write δ. Δ may be known or unknown. For our algorithms in this paper, we assume that Δ is known. However, δ is unknown (it represents the performance metric of a single *run*).

2.1 Repeated consensus

We focus on the repeated consensus problem. The in-queue and out-queue are queues of pairs $\langle i, v\rangle$, where i is a consensus instance number and v a value. In the repeated consensus problem, for each instance i, the following holds:

[2] *Validity:* A message m that is received by q was previously sent by some process p to q; *Integrity:* A message m that is sent from p to q is received by q at most once.

- *Validity*: For every process p, if $\langle i, v \rangle \in Out_p$ then there exists some process q such that $\langle i, v \rangle \in In_q$.
- *Uniform agreement:* For all processes p, q, if $\langle i, v \rangle \in Out_p$ and $\langle i, v' \rangle \in Out_q$ then $v = v'$.
- *Termination:* For all correct process p there exists v such that $\langle i, v \rangle \in Out_p$.

2.2 Swift algorithms

Before giving a formal definition of swift, we need to formalize the notion of execution time of an instance.

Definition 3 (Execution time)**.** *Consider a run R of a repeated consensus algorithm. The execution time $\tau_i(R)$ of instance i of consensus is defined as follows. Let $t_{in} = \max\{t : \langle i, v \rangle$ is taken from In_i at some process p at time $t\}$, $t_{out} = \max\{t : \langle i, v \rangle$ is output to Out_i at some process p at time $t\}$. Then $\tau_i(R) = t_{out} - t_{in}$.*

Let $A(\Delta)$ denote algorithm A parameterized with Δ.[3]

Definition 4 (Swift algorithm)**.** *An algorithm $A(\Delta)$ that solves repeated consensus is* swift *if there are constants $k, c \in \mathbb{N}$ such that for every run R of $A(\Delta)$ that is Δ-partial synchronous with good period R', there exists i' such that for all instance $i \geq i'$, we have $\tau_i(R) \leq k\delta(R') + c$.*

Note that this definition does not refer to timeouts. Timeout expiration is a low level issue. Our definition only depends on the relation between system properties (namely, transmission delays) and algorithm properties (namely, execution time), and therefore avoids any reference to timeout expirations.

3 A non-swift round-based algorithm

We illustrate swiftness and non-swiftness on simple consensus algorithms. The algorithms we consider belong all to the same class of consensus algorithms, i.e., algorithms that require $f < n/3$. In this section we consider a round-based algorithm, namely the OneThirdRule (OTR) consensus algorithm from [4], see Algorithm 1. The round-based model has been introduced in [6]. In each round r, a process sends its estimate x_p to all processes (line 6) and then, after an implicit receive step where only messages of round r may be received, performs the state transition function T_p^r (lines 8 to 11). Algorithm 1 is always safe. For liveness, we need two rounds in which the set Π_0 of alive processes (at least $2n/3$) receives all messages from processes in Π_0, and only from these processes. This property is called *space uniformity*. It can be ensured by the round implementation layer during the good period of a partially synchronous system.

[3] For models with known bounds on transmission delays, Δ represent this knowledge. For models with unknown Δ, or asynchronous algorithms, we assume $A(\Delta)$ to be a constant function, *i.e.*, $A(\Delta)$ represents one single algorithm.

Algorithm 1 OneThirdRule (OTR) (code of process p)

1: **State:**
2: $x_p \in V$
3: $decision_p \in V$

4: **Round** r**:**
5: S_p^r:
6: send $\langle x_p \rangle$ to all processes
7: T_p^r:
8: **if** number of values received $> 2n/3$ **then**
9: $x_p \leftarrow$ smallest most often received value
10: **if** more than $2n/3$ values received are equal to v **then**
11: $decision_p \leftarrow v$

Algorithm 2 A non-swift round implementation (code of p)

1: $r_p \leftarrow 1$ /* round number */
2: $next_r_p \leftarrow 1$
3: $Rcv_p \leftarrow \emptyset$ /* set of received messages */
4: $\forall i \in \mathbb{N}: \ state_p[i] \leftarrow \bot$ /* state of instance i */

5: **while** `true` **do**

input & send:
6: $I \leftarrow input()$
7: **for all** $\langle i, v \rangle \in I$ **do**
8: $state_p[i] \leftarrow \langle v, \bot \rangle$
9: **for all** $i : \ state_p[i] \neq \bot$ **do**
10: $msgs[i] \leftarrow S_p^{r_p}(state_p[i])$
11: **for all** $q \in \Pi$ **do**
12: $M_q \leftarrow \{\langle i, msgs[i][q] \rangle : \ state_p[i] \neq \bot\}$
13: send(M_q, r_p, p) to q

receive:
14: $i_p \leftarrow 0$
15: **while** $next_r_p = r_p$ **do**
16: $i_p \leftarrow i_p + 1$
17: **if** $i_p \geq TO$ **then**
18: $next_r_p \leftarrow r_p + 1$
19: receive(M)
20: $Rcv_p \leftarrow Rcv_p \cup M$
21: $next_r_p \leftarrow \max(\{r : \langle -, r, - \rangle \in Rcv_p\} \cup \{next_r_p\})$

comp. & output:
22: $O \leftarrow \emptyset$
23: **for all** $i : \ state_p[i] \neq \bot$ **do**
24: **for all** $r \in [r_p, next_r_p - 1]$ **do**
25: $\forall q \in \Pi : \ M_r[q] \leftarrow m$ if $\exists M \ \langle M, r, q \rangle \in Rcv_p \wedge \langle i, m \rangle \in M$, else $\bot$
26: $state_p[i] \leftarrow T_p^r(state_p[i], M_r)$
27: **if** the first time $state_p[i].decision \neq \bot$ **then**
28: $O \leftarrow O \cup \langle i, state_p[i].decision \rangle$
29: $output(O)$
30: $r_p \leftarrow next_r_p$

The implementation of the round structure is given by Algorithm 2. It is an extension of the implementation given in [9] with support for repeated instances of consensus.

Each iteration of the outermost loop is composed of three parts: *input & send* part, *receive* part and *comp. & output* part. In the *input & send* part, the process queries the input queue for new proposals (line 6), initializes new slots on the *state* vector for each new proposal (line 8), calls the send function of all active consensus instances (line 10), and sends the resulting messages (line 13). The process then starts the *receive* part, where it waits for messages until ei-

ther the timeout TO expires (line 17) or it receives a message from a higher round (line 21). Finally, in the *comp. & output* part, the process calls the state transition function of each active instance (line 26), and outputs any new decisions (line 29). Note that some rounds may be partially skipped (no message sent, no message received, only transition function executed): this happens whenever a message from higher round is received.

In [2] we prove the correctness of the round implementation for $TO \geq 2\Delta + 2n + 5$. We also show that for each instance i of consensus started after GST, we have an execution time $\tau_i \leq 2TO + \delta + 3n + 6$. This defines the maximum execution time. We now show that the implementation is not swift by computing the minimum execution time for each instance of consensus.

Lemma 1. *Consider Algorithm 2 with $TO \geq 2\Delta + 2n + 5$, $n > 3f$. Let R be a Δ-partial synchronous run. Let r_0 be the first new round that is started after GST. Then for all instances i started in a round $r \geq r_0$, we have an execution time $\tau_i > \Delta$.*

Proof. Assume by contradiction that for an instance i that started in a round $r \geq r_0$, we have $\tau_i \leq \Delta$. This means that there is a process p that stays in round r at most Δ time.

Let t_r and t_e be the time when p starts and finishes round r, respectively. According to the code of algorithm, process p may finish round r in two cases: either (i) by the expiration of its timeout, or (ii) by receiving a higher round message. In case (i) we have $t_e - t_r = TO + (n + 2) > \Delta$. In case (ii) let q be the first process that has finished round r and sent round $r + 1$ messages to all. Process q could do this only by expiration of its timeout for round r. Therefore, q has started round r the latest by $t_q = t_e - TO - (n + 2)$. Process q sent a round r message to p by $t_q + n + 1$, and p received it $\delta + n + 2$ later. So by $t_q + \delta + 2n + 4$, p must have entered round r, therefore $t_r = t_q + \delta + 2n + 4$. Expanding t_r, we obtain $t_r = t_e - TO - (n+2) + \delta + 2n + 4 < t_e - \Delta$, that is, $t_e > t_r + \Delta$, which contradicts the assumption that p remained in round r for at most Δ. Therefore, for all $r \geq r_0$, all processes remain in round r for more than Δ time. A contradiction. □

Since the execution time is proportional to the parameter Δ and independent of the effective transmission delay δ, the implementation is not swift:

Theorem 1. *The round implementation of Algorithm 2 is not swift.*

Proof. In case that $TO < 2\Delta + 2n + 5$, the algorithm is not live. Therefore we only consider $TO \geq 2\Delta + 2n + 5$.

Assume by contradiction that the collection of algorithms $A(\Delta)$ given by Algorithm 2 is swift. Then, there are $k, c \in \mathbb{N}$, such that in every (Δ, ∞)-partial synchronous run R with a good period R', there is an i_R such that, for all instances $i > i_R$, $\tau_i(R) < k\delta(R') + c$. For contradiction, consider $A(k + c)$. Then, for runs where $\delta(R) = 1$, by Lemma 1, there is an i_R, such that for $i > i_R$, $\tau_i > \Delta \geq k + c$, a contradiction. □

Algorithm 3 OTR with the failure detector $\Diamond\mathcal{P}$ (code of p)

```
1:  State:
2:     r_p ← 1                                          /* round number */
3:     x_p ∈ V
4:     decision_p ∈ V

5:  while true do
6:     send ⟨r_p, x_p⟩ to all processes
7:     wait until received values for round r_p from all processes q ∉ ◇P_p

8:     if number of values received > 2n/3 then
9:        x_p ← x smallest most often received value
10:       if more than 2n/3 values received are equal to v then
11:          decision_p ← v
12:    r_p ← r_p + 1
```

4 A failure detector-based algorithm that is swift

We consider now the OTR algorithm expressed with the failure detector $\Diamond\mathcal{P}$, see Algorithm 3.

Intuitively it is easy to see that repeated execution of this algorithm is swift. Indeed, some time after GST, the failure detector list contains exactly the faulty processes. At this point, by line 7, all correct processes wait only for messages from correct processes, and since $f < n/3$, the condition of line 8 is always true. Note that the failure detector model requires reliable links, contrary to the solution in the previous section.[4] For simplicity we will assume that links are natively reliable in this section, although they can be implemented from eventual reliable links with a retransmission protocol.

Repeated execution of Algorithm 3 is expressed by Algorithm 4. The box in Algorithm 4 corresponds to line 7 of Algorithm 3. For simplicity, we have not shown in Algorithm 4 the (trivial) implementation of $\Diamond\mathcal{P}$. We assume that, both Algorithm 4 and implementation of $\Diamond\mathcal{P}$, run in the same partial synchronous system in the following way: in every even step, Algorithm 4 is executed, in every odd step, the implementation of $\Diamond\mathcal{P}$ is executed.

The correctness of Algorithm 4 follows from the following lemma:

Lemma 2. *For Algorithm 4, there is eventually a round GSR so that for all rounds $r \geq GSR$, every correct pro-*

[4] Consider two correct processes p and q and line 7 executed by p. If the message sent by q is lost, and p's failure detector never suspects q, then p is blocked forever at line 7.

Algorithm 4 Multiple instances of Algorithm 3 (code of p)

```
1:  Initialization:
2:     r_p ← 1
3:     ∀i ∈ ℕ : x_p[i] ← ⊥
4:     ∀i ∈ ℕ : decision_p[i] ← ⊥

5:  while true do
6:     I ← input()
7:     for all ⟨i, v⟩ ∈ I do
8:        x_p[i] ← v
9:     send ⟨r_p, x_p, p⟩ to all processes
10:    while not received ⟨r_p, x_q, q⟩ from all processes q ∉ ◇P_p do
11:       receive(M)
12:       Rcv ← Rcv ∪ M
13:    O ← ∅
14:    for all i : x_p[i] ≠ ⊥ and decision_p[i] = ⊥ do
15:       if number of values received ⟨r_p, x′, −⟩ > 2n/3 then
16:          x_p[i] ← smallest most often value x′[i]
17:          if more than 2n/3 values x′[i] are equal to v then
18:             decision_p[i] ← v
19:             O ← O ∪ {⟨i, v⟩}
20:    output(O)
21:    r_p ← r_p + 1
```

cess receives a message from every correct process in round r and receives no message from faulty processes.

Algorithm 4 is swift as eventually every instance of consensus decides within $3\delta + 6n + 6$, see Theorem 2.

Theorem 2. *For a run of Algorithm 4 with $n > 3f$ and an infinite number of instances, there is an instance number i such that for all $i' \geq i$, we have an execution time $\tau_{i'} \leq 3\delta + 6n + 6$.*

Proof. Since in every input step only a finite number of instances are read, there is an input step so that this step and all later input steps are just before a round that is after GSR (see Lemma 2). Let i_0 be the maximum over all instance numbers that were issued before GSR. Then all instances $i' > i_0$ are started after GSR (an instance is started in the round when the last process starts that instance). In this case, the algorithm decides in at most two rounds.

It remains to calculate the maximum time for two rounds after GSR. Let t be the first time a process starts round r, let p be that process. Then every other process does so the latest at $t + 2(n-1) + \delta$ (note that we have to double the time for a step, since only every second step is of the asynchronous algorithm). To see this, note that every process sends its message for round $r-1$ the latest by $t + 2(n-1)$, since p received already this message from every correct process at this time. All other processes are performing only receive steps at this time, thus these messages are received by time $t + 2(n-1) + \delta$ and all processes are in round r.

By $t + 2\delta + 2(2n)$ all round r messages are thus ready for reception, and received by $t + 2\delta + 2(2n+1)$. Again by $t + 2\delta + 2(3n+2)$ all round $r+1$ messages are sent, and thus round $r+1$ ends the latest at $t + 3\delta + 2(3n+3)$. ☐

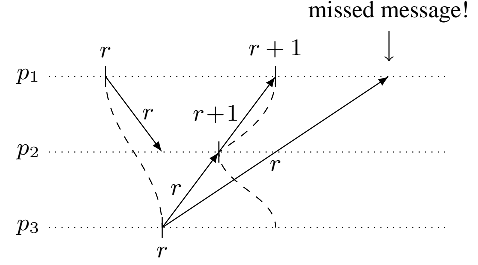

Figure 1. New round implementation: issue to address

Remark: Failure detector based solutions require reliable links. This has the following implication. In contrast to partial round implementation of Section 3, no round is skipped, *i.e.*, processes send messages for all rounds, and wait for the messages from all unsuspected processes. This implies that, unlike the round implementation in the previous section, it is no more possible to bound the time from GST until the first decision. To see this, note that at GST, a process p might be in a round r that is arbitrarily smaller than the highest round number r_{max} at that time. Since other correct processes might wait in any round r', $r \leq r' \leq r_{max}$ for the round r message of process p, p cannot skip the sending of all rounds between r and r_{max}. This can also not be easily solved by packing all messages into a single one, since between the sends p also has to perform receive steps to receive messages from the other correct processes. This takes an unbounded amount of time, as $r_{max} - r$ can be arbitrarily large.

5 A new round implementation that is swift

We show now that the implementation of the round model can be made swift. Like in the failure detector approach, each process estimates a set of alive processes (the complementary of the set of suspected processes) and uses this set to terminate a round earlier after GST, namely, as soon as it receives all messages from the alive set. Contrary to the failure detector approach, the algorithm tolerates message loss, by using a timeout which expires only before GST. Like in the round-based implementation, processes resynchronize after message-loss by skipping rounds. Skipping rounds also allows the algorithm to decide in a bounded time after GST.

5.1 Issue to address

Combining the termination of a round upon reception of all messages from alive processes, and the round-skipping mechanism, requires some attention. The problem is illustrated in Figure 1. In this scenario, p_3's round r message is the last message needed by p_2 to have all round r messages. Let's assume that upon receiving this message, p_2

Algorithm 5 A swift round implementation (code of p)

```
1:  r_p ← 1                                              /* round number */
2:  next_r_p ← 1
3:  Rcv_p ← ∅                                 /* set of received messages */
4:  ∀i ∈ ℕ : state_p[i] ← ⊥                      /* state for instance i */

5:  while true do
6:    input & send                       /* lines 6-13 of Algorithm 2 */
      ---- receive ----
7:    i_p ← 0;
8:    timeout_p ← TO
9:    while next_r_p = r_p do
10:     i_p ← i_p + 1
11:     receive(M)
12:     Rcv_p ← Rcv_p ∪ M
13:     Alive_p ← {set of processes from whom
               there is a message within last TO_A steps}
14:     if ∀q ∈ Alive_p : ∃⟨M_q, r_p, q⟩ ∈ Rcv_p then
15:       next_r_p ← r_p + 1
16:     if i_p ≥ timeout_p then
17:       next_r_p ← r_p + 1
18:     r ← max{r : ⟨−, r, −⟩ ∈ Rcv_p}
19:     if r > r_p + 1 then
20:       next_r_p ← r
21:     if there is a message from round r_p + 1
        for the first time then
22:       timeout_p ← min{i_p + TO_D, TO}
      ---- end receive ----
23:   comp. & output                    /* lines 22-29 of Algorithm 2 */
24:   r_p ← next_r_p
```

immediately sends its round $r + 1$ message to all. In this case, process p_1 may receive the round $r + 1$ message of p_2 before the round r message of p_3. If p_1 jumps to round $r + 1$ upon receiving the first round $r + 1$ message, it will miss p_3's round r message, thereby breaking space uniformity on round r. This situation may repeat in every round, thus preventing the algorithm from deciding. In Section 5.2 we show how we address this problem.

5.2 The full algorithm

The ideas described above are used in Algorithm 5, which is a round implementation that is swift. Algorithm 5 enhances Algorithm 2 as follows:

(i) Each process p maintains an estimation of the set of alive processes in $Alive_p$ (see line 13), and updates it every TO_A steps. TO_A is thus the timeout used to suspect faulty processes.

(ii) A process goes directly to the next round if it received a message from all processes in its alive set (lines 14-15). This is the key point to make the algorithm swift.

(iii) In any case, a process goes to the next round after TO time (lines 16-17). TO is thus the timeout for a round in bad periods.

(iv) When receiving a round message from the next round for the first time, the process waits for at most TO_D steps before going into this round (lines 21-22). For this and the last point, each process p maintains a variable $timeout_p$, initially set to TO (line 8) which is modified when a round $r + 1$ message is received (line 22). This is used to address the problem described in Section 5.1.

(v) When receiving a message from a round higher than the next round (*i.e.*, $> r_p+1$), the process immediately goes to this round (lines 19-20). This ensures a fast resynchronization of the processes after a bad period.

We now show correctness of this solution (Section 5.3), and then the swiftness (Section 5.4).

5.3 Correctness

Algorithm 1 together with Algorithm 5 solves repeated consensus in a partial synchronous system. As already discussed, Algorithm 1 is always safe (with $n > 3f$). Before proving that the round implementation given by Algorithm 5 provides liveness, we show some properties of the algorithm related to the correctness that hold after GST.

When the good period starts at GST, processes will synchronize to the same round using the following two mechanisms: (i) when a process receives a higher round message, it advances rounds either immediately (line 20), or within TO_D (lines 21-22), or when the original timeout TO expires; (ii) in any case, processes remain in a round at most TO time, starting a new round when this timeout expires (lines 16-17 and line 22). Therefore, shortly after GST, there will be a process p that starts a new round r that is higher than any round started by the other alive processes. When the other processes receive the round r message from p, they will advance to round r and send their own messages. These messages are then received by all alive processes, resulting in a space uniform round.

As discussed in Section 5.1, a round $r + 1$ message may be received before all round r messages (Figure 1). To address this issue, if a process p in round r receives a message from round $r + 1$ for the first time and it has not received all the messages from its alive set, it does not advance immediately. Instead, it waits either for an additional TO_D or until the end of the original timeout, whichever comes first. During the good period, all the remaining round r messages will be received before this revised timeout expires. To see why, notice that for a process to send a round $r + 1$ message, it must have received all round r messages from the alive processes, so these messages will also be received by process p within at most $TO_D = \Delta + (n - 1)$, namely $(n - 1)$ send steps and Δ maximum transmission delay. In any case, all messages will be received before the original round timeout, so the process only has to wait for the minimum of TO_D or what is left of TO.

If a process p in round r receives a message from round $r + 2$ or higher, it can conclude that the good period has not yet been started, so it advances immediately to round

$r + 2$. To see why, consider that if we are inside the good period and if a process q sends a round $r+2$ messages, then either (i) q received all round $r + 1$ messages, including p's message, which is not possible; or (ii) the timeout for round $r + 1$ expires, which is not possible as the timeout is chosen in a way that processes have enough time to receive all round messages and messages are not lost in the good period. Therefore, we are still in the bad period.

Formally, we have:

Theorem 3. *Consider a run of Algorithm 5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partial synchronous run. Then every consensus instance that starts before GST decides the latest at $GST + TO_A + TO + TO_D + 3\Delta + (5n+11)$.*

The proof is based on the following two lemmas, see [2]:

Lemma 3 (Timeouts TO and TO_D)**.** *Consider Algorithm 5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$. Let R be a Δ-partial synchronous run. Let t_r be the time the first process starts a new round r after GST. Let all processes have the same actual alive set in this interval. Then round r is space-uniform.*

Lemma 4 (Timeout TO_A)**.** *Consider Algorithm 5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partial synchronous run. Let t_r be the time the first process starts a new round r after GST. Then by time $t_r + 2 + TO_A$ all processes have the same alive set, which is the set of alive processes.*

5.4 Swiftness

In order to show that Algorithm 1 together with the round implementation provided by Algorithm 5 is swift, we show that the execution time of a consensus instance depends only on δ and not on Δ.

The main properties of the algorithm related to the swiftness, which hold after GST, are the following. First, the $Alive$ set becomes accurate the latest by $GST + TO_A$ (line 13). Then, once the alive set is accurate after GST, it no more changes and therefore no further timeout expires. Finally, all processes finish rounds as soon as all messages from alive processes are received and advance round by lines 14-15, rendering the algorithm swift.

Theorem 4. *Consider Algorithm 5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partial synchronous run. Then every consensus instance that is started after $GST + X$ with $X = TO_A + TO + TO_D + 2\delta + (4n+9)$, has an execution time of $\tau_i \leq 3\delta + (4n+7)$.*

Proof. From Lemma 4 all processes have the same alive set by $GST + TO_A$. From the code of the algorithm a process, e.g., p_1, starts a new round r every $TO + (n+2)$ steps, i.e., the latest by $t_1 = GST + TO_A + TO + (n+2)$. All processes do so by $t_2 = t_1 + TO_D + \delta + (2n+5)$. This means that all processes start round r with the same alive set. From Lemma 3, round r is space uniform. Furthermore, all processes receive all round r messages from their alive set, and end round r the latest by $t_3 = t_2 + \delta + (n+2)$ and start round $r+1$ at this time. Therefore, all processes end the first space-uniform round the latest by time $GST + TO_A + TO + TO_D + 2\delta + (4n+9)$. This proves the first part of the theorem with $X = TO_A + TO + TO_D + 2\delta + (4n+9)$.

Similar to the proof of Theorem 2, there are instances that start after $GST + X$. It remains to calculate the execution time for a consensus instance that is started after $GST + X$. Since no process crashes after GST, the alive set remains the same (see line 13). Let t be the first time process p_1 starts a new round r. Then every other processes does so $(n-1) + \delta + (n+4)$ steps later. This is because some process p_2 might send the last round $r-1$ message $(n-1)$ steps later to another process, e.g., p_3. And p_3 will start round r the latest after $(n+4)$ steps (an output followed by an input, n send, one receive, and one output step). Therefore, by $t + \delta + (2n+3)$ all processes start round r. By $t + 2\delta + (3n+4)$, all round r messages can be received, and round r ends after an output step by time $t + 2\delta + (3n+5)$. Again by $t + 2\delta + (4n+6)$ all round $r+1$ messages are sent, and thus round $r+1$ ends the latest at $t + 3\delta + (4n+7)$. □

6 Experimental results

In this section we present the results of an experimental study, comparing the three algorithms presented previously. The main questions we want to answer are (i) how much improvement can be obtained in a round-based algorithm using a swift round implementation, and (ii) are swift round implementations competitive with implementations that use failure detectors.

Experimental setup We performed our experiments both on an emulated network and directly on a physical network (a cluster). The emulated network allowed us to test the behavior of the algorithms with different transmission delays and message loss rates, while the physical network shows what to expect on a cluster environment.

In all experiments, processes were started with 1 second of delay between each other. This prevents initial synchronization and exercises the ability of the algorithms to resynchronize the processes.

The metric considered is the decision time for each consensus instance. Processes run each instance sequentially, starting the next one either when they decide or learn the decision by receiving a message from a higher instance. To compute the execution time, we consider the time between the moment the first process starts instance i until the first decision. Each data point shown on the plots below was obtained from a 10 minutes run. We then calculated the average decision time, ignoring the first 10% of the run. For each data point, we show the 95% confidence intervals.

Implementing $\Diamond\mathcal{P}$ and reliable channels for the failure detector algorithm We implemented $\Diamond\mathcal{P}$ by having each process send heartbeats to all every η time. A process p suspects q if it does not receive any heartbeat for more than τ time. We also implemented reliable channels using message acknowledgments and retransmission. We decided not to use TCP, because our initial experiments using it resulted in a very poor performance under high message loss conditions. TCP is designed to interpret message loss as an indication of congestion, and therefore it reacts by increasing the retransmission time. On a typical TCP implementation, the interval between retransmissions may reach several minutes, which in practice forces the algorithms running on top of it to stop.

Notation In the following, δ_{net} denotes the one-way transmission delay of the physical network, δ_{emu} the delay emulated by ModelNet, and δ_{eff} the effective one-way transmission delay between two processes. On the experiments run directly on the physical network, $\delta_{\mathit{eff}} = \delta_{net}$. However, when using ModelNet, $\delta_{\mathit{eff}} = 2\delta_{net} + \delta_{emu}$, since the packet is transmitted two times on the physical network between two nodes of the ModelNet. Finally, note that contrary to δ defined previously in the paper, δ_{net} is not a bound. Instead, it is a random variable, reflecting the non-deterministic behavior of a physical network.

In the following, NS-OTR, S-OTR, and FD-OTR denote the non-swift OTR (Algorithm 1 + Algorithm 2), swift OTR (Algorithm 1 + Algorithm 5) and OTR with FD (Algorithm 4 + $\Diamond\mathcal{P}$), respectively.

6.1 Emulated network

We used ModelNet [13] to emulate a network. ModelNet uses two types of nodes: a *core node* that applies the traffic policies, and one or more *edge nodes* that run the application being tested. The edge nodes redirect all traffic sent by the processes to the core node, which applies the traffic policy (*e.g.*, delay, loss and maximum bandwidth) and then transmits the packet to the intended receiver. We varied the emulated delay and loss rate, while leaving the emulated bandwidth set to 1Gbps. We used two physical machines for all experiments run on ModelNet. All 4 replicas were running on a dual Pentium 4 at 3.6GHz with 1GB RAM, while the core node was a Pentium Pro at 200MHz with 70MB of RAM. The machines were connected by a full duplex 100Mbits Ethernet, and had a ping time of ≈ 0.3ms. Hence, $\delta_{\mathit{eff}} \approx 0.3 + \delta_{emu}$.

Varying the timeout In the first set of experiments, we fixed the emulated transmission delay while varying the timeout (TO) used by the algorithms.

Figure 2 shows the results for $\delta_{emu} = 0ms$ and Figure 3 for $\delta_{emu} = 40ms$. The x scale indicates the timeout (TO) used by the algorithms to terminate a round.[5] For the tests with $\delta_{emu} = 40ms$, the failure detector was configured with $\eta = TO/2$ and $\tau = TO$. The rationale is that TO is the time an algorithm should wait before declaring a failure and taking corrective measures, *e.g.*, advancing rounds or suspecting a process. With $\delta_{emu} = 0ms$, following the same policy would result in the network being overloaded with heartbeats, so we opted for $\eta = TO$ and $\tau = TO * 2$.

The results clearly validate the main motivation behind this work, in that S-OTR performs at the speed of the network, being independent from the timeout.

With $\delta_{emu} = 0ms$ (Figure 2-left), FD-OTR performs poorly with low timeouts. This is caused by the additional messages sent by the failure detector and the reliable channels implementation, which slow down the processes and congest the network. For higher timeouts, this overhead becomes less significant and the algorithm starts performing similarly to the other implementations. When looking only at NS-OTR vs S-OTR (Figure 2-right), it is clear that the decision time of NS-OTR increases linearly with the timeout, while the swift OTR is constant. Furthermore, even with the optimal timeout of $2ms$, the NS-OTR performs worse than S-OTR.

With $\delta_{emu} = 40ms$ (Figure 3-left), NS-OTR performs poorly with timeouts lower than $80ms$. For timeouts lower than $60ms$, it is hardly able to decide even a few times, so we did not show the results as they were not statistically significant. Notice that $80ms \approx 2\delta_{\mathit{eff}}$, which matches the results from the analytical analysis, where a round must last $TO = 2\Delta$ in order to ensure decision. The swift version is more tolerant to a non-optimal timeout, being able to synchronize even with timeouts of a little above $40ms$. This is because processes finish rounds early after receiving all messages, allowing the processes that are most behind to slowly catch-up with the ones in the lead.

FD-OTR is also independent of the timeout, producing the optimal performance regardless of the values used for the underlying failure detector. Recall that in the absence of message loss, the values chosen for the failure detector (*i.e.*, $\tau = 2\eta$) ensure that there will be no false suspicions,

[5] Equivalent to 2Δ on the NS-OTR and 3Δ for the S-OTR.

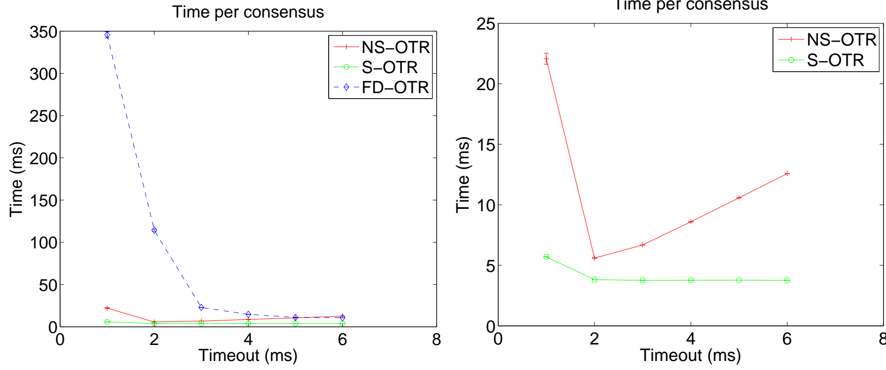

Figure 2. Performance on ModelNet with $\delta_{eff} \approx 0.3ms$ ($\delta_{emu} = 0, 2\delta_{net} \approx 0.3$)

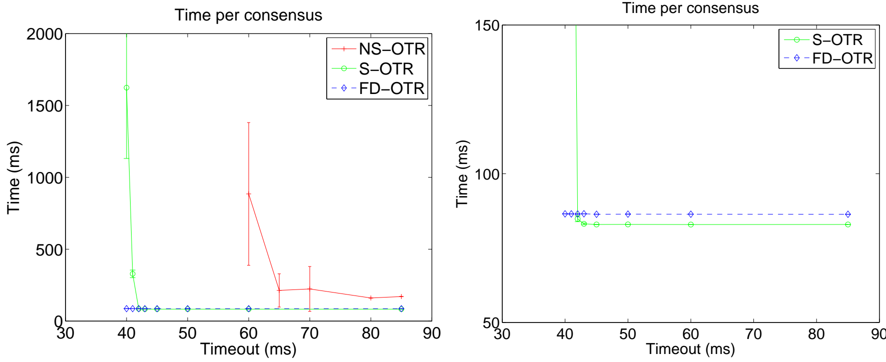

Figure 3. Performance on ModelNet with $\delta_{eff} \approx 40.3ms$ ($\delta_{emu} = 40, 2\delta_{net} \approx 0.3$)

so FD-OTR can proceed at the speed of the network. The overhead of the failure detector and of the reliable channels is less in this scenario, as noticeable in Figure 3-right, where it performs only slightly worse than S-OTR.

Message loss Figure 4 shows the behavior of the algorithms in networks with message loss. The experiment was run on ModelNet with $\delta_{emu} = 0$. Both swift and the non-swift versions were configured with a timeout of $10ms$. The failure detector was configured with $\eta = 10ms$ and $\tau = 25ms$, so that it tolerates 2 or 3 lost heartbeats before (wrongly) suspecting a process. The reliable channels implementation retransmits a message every $25ms$.

Both the NS-OTR and S-OTR are very resilient to message loss. Even with 40% messages loss, the average decision time is only a few milliseconds more than when there is no message loss. This is because the algorithm makes progress as soon as a single process receives 3 messages ($2n/3$), *i.e.*, 2 messages from other processes since its own message is always delivered.

S-OTR outperforms both NS-OTR and FD-OTR in the presence of message loss. In particular, the performance of FD-OTR degrades a lot with message loss, caused by the overhead of the retransmissions to simulate reliable links.

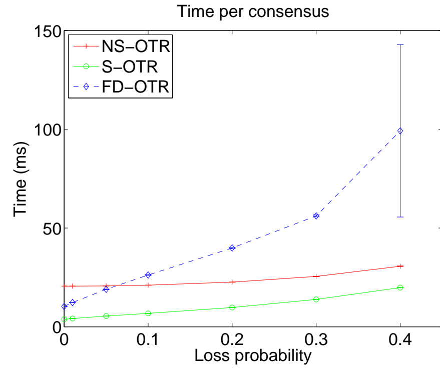

Figure 4. Performance with message loss: $\delta_{eff} \approx 0.3ms$ ($2\delta_{net} \approx 0.3, \delta_{emu} = 0$)

6.2 Physical network (Cluster)

For the tests with the physical network, we used a cluster of Dual Pentium 4 at 3.00GHz with 1GB memory connected by a 1Gbit Ethernet. Each process run on a separate

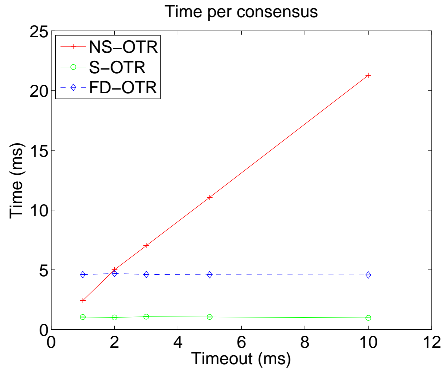

Figure 5. Performance on cluster ($\delta_{eff} \approx 0.1ms$)

node and the ping time between two nodes was between 0.1 to 0.2ms. The failure detector was configured with $\eta = TO$ and $\tau = TO * 2$.

Figure 5 shows that on the cluster even a timeout of $1ms$ is enough for OTR to terminate. S-OTR always outperforms the two other algorithms. Compared to NS-OTR, even with a $1ms$ timeout, S-OTR performs better. Lowering the timeout of the non-swift version even further may improve its performance. But at such small timeouts, the algorithm becomes sensible to the normal variability of the system, which is caused by non-deterministic factors like OS scheduling or background activity, either on the hosts or on the network. This will cause rounds to finish without receiving all required messages, leading to unstable performance. The timeout of S-OTR can be set to a conservative value, making the algorithm immune to non-deterministic factors, while still providing optimal performance.

FD-OTR suffers again from the overhead of the underlying failure detector and of the reliable channels, resulting in a worst performance than S-OTR.

7 Discussion

Table 1 summarizes the results of the paper. We have analyzed efficiency of algorithms in two models for solving consensus: the round-based model (which can be implemented on top of a partially synchronous system), and the asynchronous system augmented with failure detectors. Efficiency refers here to *swiftness*, a new notion that captures the fact that an algorithm, once the system has stabilized, progresses at the speed of the messages. Our new round-based implementation combines the advantages of failure detectors solutions (swiftness) and round-based model (lossy links). This weak link assumption makes round-based algorithm easy to adapt to the crash-recovery model with stable storage [9].

We have illustrated the new round-based implementation on a specific consensus algorithm (OTR). This does not mean that the new solution is limited to OTR. It applies to any consensus algorithm expressed in the round model, in particular to the *LastVoting* algorithm [4], a round-based variant of Paxos [11] that requires only $n > 2f$.

Algorithms	Simple round-based [9] (Algorithm 2)	FD-based [8] (Algorithm 4)	New round-based (Algorithm 5)
Link	lossy	reliable	lossy
Execution time	$4\Delta + \delta + O(1)$	$3\delta + O(1)$	$3\delta + O(1)$
Swift	no	yes	yes

Table 1. Repeated consensus: algorithms analyzed in the paper

References

[1] M. Biely and J. Widder. Optimal message-driven implementations of omega with mute processes. *ACM Trans. Auton. Adapt. Syst.*, 4(1):1–22, 2009.

[2] F. Borran, M. Hutle, N. Santos, and A. Schiper. Swift Algorithms for Repeated Consensus. Technical report, EPFL, 2009. `http://infoscience.epfl.ch/record/142723`.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

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

[5] C. Delporte-Gallet, S. Devismes, H. Fauconnier, F. Petit, and S. Toueg. With finite memory consensus is easier than reliable broadcast. In *OPODIS*, pages 41–57, 2008.

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

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

[8] E. Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *PODC'98*, pages 143–152, Puerto Vallarta, Mexico, 1998. ACM Press.

[9] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *DSN 2007*, pages 92–10. IEEE, June 2007.

[10] M. Hutle and J. Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Self-Stabilizing Systems*, pages 153–170, 2005. Appeared also as Brief Announcement at PODC'05.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[12] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[13] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.