Stubborn Communication Channels

Rachid Guerraoui Rui Oliveira André Schiper

Départment d'Informatique

Ecole Polytechnique Fédérale de Lausanne

Lausanne, Switzerland

Abstract

This paper aims at bridging the gap between the assumption of reliable channels by fault-tolerant distributed algorithms and the weak reliability of feasible communication channels. We define a new kind of communication channels which we call *Stubborn* channels. Stubborn channels are easily implementable over a connectionless network layer and, although weak, the reliability guarantees offered by Stubborn channels are sufficient to solve the fundamental Consensus problem in asynchronous systems with unreliable failure detectors.

1 Introduction

Literature on fault-tolerant distributed algorithms in the asynchronous computation model often assumes that the communication between processes is reliable. A reliable communication channel can roughly be characterized as a channel where all messages that are sent are received, provided that the receiver process does not crash [2]. In practice, implementation of reliable channels poses two problems. The first problem is to guarantee the eventual reception of a message, in case the sender crashes after having sent the message. The second problem is the buffer space required at the sender process: in an asynchronous system where a crashed process is indistinguishable from a slow process or from a process connected through a slow channel, implementing reliable channels requires unbounded buffer space. Due to these limitations, reliable channels are only approximated in practice, and almost invariably by *best-effort* channels. Best-effort channels, of which TCP/IP [5] is a well-known example, have weaker semantics than reliable channels, because no delivery guarantee is provided if the sender either crashes, or suspects the receiver of having crashed (even though the latter has not crashed).

Our study aims at bridging the gap between the weak reliability of feasible communication channels (i.e., best-effort channels) and the assumption of reliable channels by fault-tolerant distributed algorithms. We seek to find, on one hand, to what extent can we sensibly improve the reliability of practical communication channels and, on the other hand, whether the reliable channel assumption is necessary in fault-tolerant distributed algorithms. We focus in particular on algorithms that solve the Consensus problem: a fundamental problem in fault-tolerant distributed computing[1]. We consider this problem in the context of asynchronous systems with unreliable failure detectors [4].

In this paper we define a new kind of communication channels which we call *Stubborn* channels. Stubborn channels do not require unbounded buffer space and are easily implementable over a connectionless network layer. Although weak, the guarantees offered by Stubborn channels allow us to present an algorithm that solves Consensus in an asynchronous system with unreliable failure detectors, releasing the assumption of reliable channels.

The rest of the paper is structured as follows. In Section 2 we discuss the reliability and feasibility of communication channels. In Section 3 we define Stubborn channels and present an implementation. In Section 4 we recall the Consensus problem, we present an algorithm that solves Consensus with Stubborn channels, and we sketch the algorithm's proof of correctness. In Section 5 we discuss related work by pointing out the practical impact of our solution. Finally, in Section 6 we summarize the paper contributions.

2 Communication channels

A reliable communication channel is a useful abstraction when designing and proving the correctness of distributed algorithms. However, in practice, the provision of reliable channels, if not impossible, is rather problematic. Practical commonly available transport protocols reflect this difficulty by providing, at most, best-effort semantics. We discuss here various communication channel semantics, starting from basic unreliable communication channels, and then re-

[1] The importance of the Consensus problem is due to the fact that many practical agreement problems, e.g., total order broadcast, non-blocking weak atomic commitment and virtually synchronous communications can be solved by reduction to Consensus [4, 9, 10].

lating the definitions to the abstract reliable channels assumed in the literature.

2.1 Basic unreliable communication channels

Existing network layers are not fault-free: messages may be lost, reordered, and duplicated[2]. Basic unreliable channels can be defined with the following three properties:

Property 1 (fairness) *If a process p sends an infinite number of messages to q, and q does not crash, then q receives an infinite subset of these messages.*

Property 2 (no-creation) *If a process q receives message m, then some process p has sent m to q.*

Property 3 (no-duplication) *Every message m that is sent by any process p is received at most once.*

Property 1 is similar to the *Weak Loss Limitation* in [11] and the *Fair Lossy* channel in [2]. This property can even be satisfied with network partitions, given that they are eventually repaired. It reflects the usefulness of the channel. Without such a property (e.g., with permanent network partitioning) any interesting distributed problem would be trivially impossible to solve. Properties 2 and 3 define a communication channel similar to the *non-duplication* channel of [1] which is a natural abstraction of the service provided by a connectionless network layer.

2.2 Best-effort communication channels

Informally, best-effort channels do their best to provide the abstraction of a reliable channel, but only as long as no process or link failure is suspected. A best-effort channel can be defined as a channel that, in addition to the properties 2 and 3 above, satisfies the following property:

Property 4 (best-effort) *If process p sends a message m to process q, and neither p suspects q, nor q suspects p, then q eventually receives m.*

[2] Integrity faults such as corrupted or randomly created messages may also occur. However, the integrity of received messages can be easily checked through checksum techniques which transforms integrity faults to loss faults.

This property introduces the notion of *failure suspicion*. A failure suspicion can be the result of an expired time-out, which can reflect either (1) the crash of the other process, or (2) a link failure. The expired time-out can also be the result of (3) a (temporary) slow process, or (4) a (temporary) slow link. Best-effort channels implement message retransmission to overcome message loss. When a failure suspicion occurs, no reliability is provided by best-effort channels (retransmission of not acknowledged messages stops), which means that the degree of reliability is directly dependent on the failure suspicion mechanism. The well-known TCP protocol is a typical example of an implementation of best-effort channels[3]. TCP is a connection-oriented protocol, and the notion of suspicion corresponds to the loss of the connection. Reestablishing the connection is not sufficient: the sender has no information about the messages that have been lost, i.e., the sender does not know what messages sent before the loss of the connection have to be retransmitted.

From a conceptual point of view, best-effort channels are too weak for solving agreement-like problems, e.g. Consensus, with the weakest knowledge about process failures. For instance, while with reliable channels the Consensus algorithm proposed by Chandra and Toueg [4] only requires that eventually some correct process p is not suspected by any other correct process, with best-effort channels it would also be required that eventually no other correct process is suspected by p (in order for these processes to receive p's decision). In Section 3, the additional semantics required by best-effort channels in order to solve such a problem with the weakest knowledge about process failures will be made clear.

2.3 CR-reliable channels

Reliable channels ensure that no message sent to a correct process is lost. A reliable channel is a channel that, in addition to the properties 2 and 3 above, satisfies the following property:

Property 5 (reliability) *If process p sends a message m to process q, and q does not crash, then q eventually receives m.*

A reliable channel ensures that a message m sent by a process p to q is received by q if q does not crash, even if p crashes immediately after having

[3]TCP provides additionally FIFO semantics, but this is not relevant to our discussion.

sent m. This is a strong requirement. The Chandra-Toueg algorithms that solve Consensus with unreliable failure detectors [4] are also correct with a weaker form of reliable channels, which we call *Correct-Restricted* reliable channels (or simply CR-reliable channels). A CR-reliable channel is a channel that, instead of the Property 5 above, satisfies the following property:

Property 6 (CR-reliability) *If process p sends a message m to process q, and p and q do not crash, then q eventually receives m.*

Reliable and CR-reliable channels are equivalent as long as the sender process p does not crash. CR-reliable channels do not provide any guarantee if the sender p crashes. CR-channels can be implemented over basic unreliable channels by retransmission. Contrary to the best-effort channels, where no guarantee is provided when suspicions occur (because retransmission stops in case of suspicions), implementation of CR-reliable channels requires retransmission until an acknowledgment is received. This can last for an unbounded duration, which is problematic from a practical point of view. Consider a process p continuously sending new messages to a process q. The number of messages that have to be stored by p (for retransmission) grows continuously as long as p does not receive acknowledgments from q. This situation can occur either because (1) q has crashed, (2) the link between p and q is down, (3) q is very slow, or (4) the link between p and q is very slow. From an abstract point of view, this means that the implementation of CR-reliable channels requires unbounded buffer space. Eliminating this assumption, which is not realistic in practice, has led us to define the notion of Stubborn channels.

3 Stubborn communication channels

In the previous section we discussed the problems of providing reliable communication channels in practice. We have considered desirable CR-reliable channels and existing best-effort channels. We define here a new kind of communication channels, called Stubborn, which are weaker than CR-reliable channels, but stronger than best-effort channels. Then we show how these channels can be implemented over basic unreliable channels. In Section 4 we give an algorithm that solves Consensus with unreliable failure detectors and Stubborn channels.

3.1 Definition

Stubborn communication channels establish a tradeoff between CR-reliable and best-effort channels. Unlike CR-reliable channels, Stubborn channels are bounded in buffer space as long as the size of the messages is bounded, and unlike best-effort channels, the reliability guarantees of Stubborn channels are independent of failure suspicion. We formally define a Stubborn communication channel between processes p and q by the two primitives, $S\text{-}send_{p,q}$ and $S\text{-}receive_{p,q}$. We say that the channel is Stubborn if, in addition to the no-creation property (Property 2 introduced in Section 2.1), it satisfies the following property.

Property 7 (Stubborn) *Let p and q be two correct processes (i.e. that do not crash). If p S-sends a message m to q, and p indefinitely delays sending any further message to q, then q eventually S-receives m.*

Basically, Property 7 says that, if the sender p (that does not crash) sends[4] message m, and afterwards is able to indefinitely delay the sending of any further message, then q eventually receives m. This does not mean that, in order to ensure the reception of m, process p is not allowed to send any new message m' to q after having sent m. This actually means that there is no fixed delay, after which, m' can be sent without compromising the reception of m. Before sending m', either m has been received or m may never be received by q. This is expressed abstractly by the indefinite delay requirement.

From Property 7, it is clear that Stubborn channels can lose messages. Therefore, Stubborn channels are weaker than CR-reliable channels. Moreover, the property does not enforce any message ordering nor prevents message duplicates.

3.2 Implementation of Stubborn channels

Figure 1 depicts a possible implementation of Stubborn channels over basic unreliable channels that provide a *send* and a *receive* primitive satisfying Properties 1, 2 and 3 introduced in Section 2.1. Ignore for now the lines starting with $*$ on the figure as the implementation does not require these lines (this code will only be useful in the next section).

[4] We should say *S-sends* to be rigorous. However, when clear from the context, we use *send* and *receive* instead of *S-send* and *S-receive*.

```
    variables
      Outbuf_{p,q} ← ⊥
      Inbuf_{p,q} ← ⊥
*     FinalSend_{p,q} ← false

    procedure S-send_{p,q}(m)
*     if not FinalSend_{p,q} then
        Outbuf_{p,q} ← m

    procedure S-receive_{p,q}(cond, m)
      m ← ⊥
      if cond(Inbuf_{p,q}) then
        m ← Inbuf_{p,q}

    task sender :
    periodically execute
      send_{p,q}(Outbuf_{p,q})

    task receiver :
    periodically execute
      receive_{p,q}(m)
      if m ≠ ⊥ then
        Inbuf_{p,q} ← m

*   procedure S-finalsend_{p,q}(m)
*     S-send_{p,q}(m)
*     FinalSend_{p,q} ← true
```

Figure 1: Implementation of Stubborn channels ($*$ lines can be ignored here)

A Stubborn channel requires buffer space for 1 message at both the sender and the receiver site ($Outbuf_{p,q}$ and $Inbuf_{p,q}$). Initially, both $Outbuf_{p,q}$ and $Inbuf_{p,q}$ are assigned $\perp$. When a process p executes $S\text{-}send_{p,q}(m)$, $Outbuf_{p,q}$ is assigned m and any previous content of $Outbuf_{p,q}$ is discarded. Periodically, $Outbuf_{p,q}$ is sent to q through the basic channel, and the basic channel is polled for new messages. If a message m is received through the basic channel, $Inbuf_{p,q}$ is assigned m and any previous content of $Inbuf_{p,q}$ is discarded.

When a process p executes $S\text{-}receive_{p,q}(cond, m)$, $Inbuf_{p,q}$ is returned if it satisfies condition *cond*. Otherwise $\perp$ is returned. No message is removed from $Inbuf_{p,q}$ and this reflects the fact that *S-receive* may return message duplicates. As we will see in Section 4, the *conditional reception* mechanism can be used by the invoking process to check for lost messages.

The implementation of Figure 1 trivially preserves the no-creation property of basic unreliable channels. To see how this implementation also satisfies the Stubborn property (Property 7), consider a message m, sent by a correct process p to a correct process q, and assume that p indefinitely delays the sending of further messages. Task *sender* periodically sends m (in $Outbuf_{p,q}$)

through the basic channel. Since the basic channel does not create nor duplicate messages (Properties 2 and 3), there is a time t after which all previous messages sent by p to q, either have been received by q, or have been lost (since the task *sender* only retransmits the latest message m). As p's last message m is periodically sent after t, by the fairness property (Property 1), m is eventually received by q. When message m is received by q, $Inbuf_{q,p}$ is assigned m. After m has been received, since p indefinitely delays any further messages to q, $Inbuf_{q,p}$ remains immutable for an indefinite long period and m is then stably available for p's *S-receive* actions.

To stress the semantics of Stubborn channels, the implementation of Figure 1 has been kept simple and does not aim at being efficient. In particular, the implementation leads to unnecessary retransmissions, which can significantly increase CPU and network loads. To reduce message retransmission, message acknowledgments can be used. Also, in periods during which the sender process p suspects the destination process q, retransmission can be temporarily stopped. Instead of implementing Stubborn channels over basic unreliable channels (eg. UDP), one could choose to add the Stubborn semantics to a best-effort communication channel, such as TCP, which already handles part of the necessary retransmission of messages and acknowledgments. *S-sending* a message m would consist in locally buffering m as the most recent message that is *S-sent*, and sending m through the TCP sending primitive. After a *connection lost* exception[5], and upon the re-establishment of the connection, the buffered message is resent through the TCP sending primitive.

4 Solving Consensus with Stubborn channels

The Consensus problem is one of the most fundamental problems in fault-tolerant distributed systems. Many agreement problems that have to be solved in fault-tolerant distributed applications are equivalent to Consensus [4, 9, 10]. We give here a Consensus algorithm based on Stubborn channels. Our algorithm is patterned after the one of Chandra and Toueg (hereafter denoted C&T algorithm) [4], which requires a majority of correct processes, reliable channels, and a failure detector of class $\diamond\mathcal{S}$ (Section 4.1). This class is proved to be the

[5] Actually, the connection lost notification of current TCP implementations does not allow callback registration which difficults their use for the implementation of Stubborn channels over TCP.

weakest[6] class of failure detectors that allow to solve Consensus [3].

Our algorithm requires a majority of correct processes, a failure detector of class $\diamond\mathcal{S}$ and Stubborn channels. Interestingly, in situations where no process is suspected and no message is lost, our algorithm reaches consensus with the same number of communication steps as the original C&T algorithm.

In the following, we define our model of computation and recall the definition of the Consensus problem, then we describe our algorithm, and sketch its proof of correctness.

4.1 System model

We consider an asynchronous distributed system composed of a finite set of processes $\Omega = \{p_1, p_2, \ldots, p_n\}$ completely connected through a set of Stubborn channels. Communication is by message passing. Asynchrony means that there is no bound on communication delays, nor on process relative speeds. Processes fail by crashing (we do not consider Byzantine failures). A correct process is a process that does not crash in an infinite run.

As in [4], we augment the asynchronous system with failure detectors. A failure detector can be viewed as a distributed oracle. Each process p_i has access to a local failure detector module D_i which maintains a list of processes that it currently *suspects* of having crashed. Failure detectors are abstractly characterized by their *completeness* and *accuracy* properties. In the following, we consider failure detectors in class $\diamond\mathcal{S}$ (*Eventually Strong*) [4]. These failure detectors guarantee (1) *strong completeness* (eventually every process that crashes is permanently suspected by every correct process), and (2) *eventual weak accuracy* (eventually some correct process is never suspected by any correct process). Failure detectors in $\diamond\mathcal{S}$ are unreliable as they can make an infinite number of false failure suspicions.

A failure detector can be implemented based on time-outs. The choice of the time-out value is system dependent. The completeness property of $\diamond\mathcal{S}$ is trivially satisfied with time-outs, whereas satisfying the accuracy property is a probabilistic issue. If a time-out is chosen too small for a given system, then the probability that the accuracy property of $\diamond\mathcal{S}$ is satisfied is close to 0. If the time-out value is chosen adequately, the probability that the accuracy property of $\diamond\mathcal{S}$ is satisfied can come close to 1.

[6] This lower bound has been stated for $\diamond\mathcal{W}$ in [3]. Similarly to [3], in our model $\diamond\mathcal{W} \cong \diamond\mathcal{S}$.

4.2 The Consensus problem

The Consensus problem (or simply Consensus) is defined over a set $\Pi \subset \Omega$ of processes. Every process $p \in \Pi$ starts with an initial value v_p, and the processes of Π have to decide on a common value v. Formally, Consensus is defined by the following properties:

Termination: Every correct process eventually decides.

Agreement. No two correct processes decide differently.

Validity. If a process decides v, then v is the initial value of some process.

4.3 The algorithm

The algorithm of Figure 2 satisfies the safety properties of Consensus, namely Agreement and Validity, even if an arbitrary number of processes crash, and the failure detector does not ensure any completeness or accuracy property. The Termination property requires a majority of correct processes and the completeness and accuracy properties of $\Diamond\mathcal{S}$.

Every correct process p starts the Consensus algorithm of Figure 2 invoking procedure *propose* with its initial value v_p. Procedure *propose* consists of two concurrent tasks: *main* and *receive_decision*. Processes communicate by means of the *S-send*, *S-receive*, and *S-finalsend* primitives introduced in Figure 1. The $S\text{-}finalsend_{p,q}$ primitive establishes the last message *S-sent* by process p to process q, that is, after executing $S\text{-}finalsend_{p,q}$, any further messages sent by p to q are discarded.

The main task

Task *main* contains the main algorithm. As in [4], this task proceeds in asynchronous rounds and uses the "rotating coordinator" paradigm. Each round has a designated coordinator which tries to reach a decision. At any time, every process, that did not crash, knows that the coordinator of round r is process $c = (r \bmod n) + 1$.

Compared to the C&T algorithm, the key characteristics of our algorithm are the changes in the **wait** statements of every phase (lines 12, 20, and 30) and the introduction of a new **wait** statement (line 26) which prevents the participants from moving to the next round if the current coordinator is not suspected. In all **wait** statements, when a process p expects a message m

procedure *propose*(vp)

$estimate_p \leftarrow v_p$ $\{p\text{'s current estimate of the decision}\}$

$r_p \leftarrow 0,\ s_p \leftarrow (0,0)$ $\{p\text{'s current round and step}\}$

$ts_p \leftarrow 0$ $\{\text{round in which } p \text{ adopted } estimate_p\}$

cobegin

$\|$ **task** *main* $\{\text{beginning of the main task}\}$

while *True*

$r_p \leftarrow r_p + 1$

$c_p \leftarrow (r_p \ \mathbf{mod}\ n) + 1$ $\{c_p \text{ is the current coordinator}\}$

$s_p \leftarrow (r_p, 1)$ $\{\textbf{phase 1}\}$

$S\text{-}send_{p,c_p}(s_p, estimate_p, ts_p)$ $\{\text{send } estimate_p \text{ to } c_p\}$

$s_p \leftarrow (r_p, 2)$ $\{\textbf{phase 2}\}$

if $p = c_p$ **then** $\{c_p \text{ gathers } \lceil\frac{(n+1)}{2}\rceil \text{ estimates, and sends a new estimate to all}\}$

wait until [for $\lceil\frac{(n+1)}{2}\rceil$ processes q : $S\text{-}receive_{p,q}(s_q \geq (r_p,1), m)$]

if [**for** $\lceil\frac{(n+1)}{2}\rceil$ **processes** q : received $m = ((r_p,1), estimate_q, ts_q)$] **then**

$msgs_p[r_p] \leftarrow \{((r_p,1), estimate_q, ts_q) \mid p \text{ received } m = ((r_p,1), estimate_q, ts_q)\}$

$t \leftarrow$ largest ts_q s.t. $((r_p,1), estimate_q, ts_q) \in msgs_p[r_p]$

$estimate_p \leftarrow$ select one $estimate_q$ s.t. $((r_p,1), estimate_q, t) \in msgs_p[r_p]$

$S\text{-}send_{p,all}(s_p, estimate_p, newestimate)$

else $S\text{-}send_{p,all}(s_p, \_, noestimate)$

$s_p \leftarrow (r_p, 3)$ $\{\textbf{phase 3}\}$

wait until [$S\text{-}receive_{p,c_p}(s_{c_p} \geq (r_p,2), m)$ **or** $c_p \in D_p$] $\{\text{all wait for new estimate}\}$

if [received $m = ((r_p,2), estimate_{c_p}, newestimate)$] **then**

$estimate_p \leftarrow estimate_{c_p}$

$ts_p \leftarrow r_p$

$S\text{-}send_{p,c_p}(s_p, ack)$

if $p \neq c_p$ **then**

wait until [$S\text{-}receive_{p,c_p}(s_{c_p} \geq (r_p,4), m)$ **or** $c_p \in D_p$] $\{\text{wait for } c_p \text{ to proceed}\}$

else $S\text{-}send_{p,c_p}(s_p, nack)$

$s_p \leftarrow (r_p, 4)$ $\{\textbf{phase 4}\}$

if $p = c_p$ **then** $\{c_p \text{ gathers } \lceil\frac{(n+1)}{2}\rceil \text{ acks, and sends decision to all}\}$

wait until [for $\lceil\frac{(n+1)}{2}\rceil$ processes q : $S\text{-}receive_{p,q}(s_q \geq (r_p,3), m)$]

if [for $\lceil\frac{(n+1)}{2}\rceil$ processes q : received $m = ((r_p,3), ack)$] **then**

$S\text{-}finalsend_{p,all}(s_p, estimate_p, decide)$

else $S\text{-}send_{p,all}(s_p, \_, nodecide)$

$\|$ **task** *receive_decision* $\{\text{beginning of the decision task}\}$

wait until [$S\text{-}receive_{p,q}(((*,4), *, decide), (s_q, estimate_q, decide))$]

$decide(estimate_q)$

$S\text{-}finalsend_{p,all}(s_q, estimate_q, decide)$

coend

Figure 2: Consensus with Stubborn channels and $\diamond\mathcal{S}$

from a process q, process p waits until m, or any subsequent message from q, is received ($S\text{-}receive_{p,q}(cond, m)$). Process p proceeds accordingly to the reception of m, and if m has not been received, p considers m lost. These differences reflect the possibilities of message loss, due either to omissions in the underlying communication channel or to a buffer overflow.

In the following, we give an operational description of the algorithm. Every round is divided into four asynchronous phases. Each process knows at any time its current *step*, where a step is a pair (round, phase). Consider a round r and a correct process p. If p is a participant ($p \neq c_p$), then p only executes phases 1 and 3. If p is the coordinator ($p = c_p$), then p executes all the 4 phases.

1. *Phase 1* (line 8). Process p sends its current $estimate_p$ of the decision value to the current coordinator, c_p. The $estimate_p$ is timestamped with the round number in which p *adopted* this estimate (see phase 3). Initially, the estimate is the initial value of the process, timestamped with 0.

2. *Phase 2* (line 10). The coordinator p ($p = c_p$) waits for estimates from a majority of processes from phase 1. From each process q of a majority of processes, p either receives $estimate_q$, or p considers $estimate_q$ lost. If p receives a majority of estimates, p selects one with the largest timestamp, and sends a *newestimate* message to all the processes as their new estimate. Otherwise, as p was unable to gather a majority of estimates, p sends a *noestimate* message to all processes and proceeds.

3. *Phase 3* (line 19). Process p waits for a message from the coordinator, c_p, from phase 2. Process p either receives the message, suspects c_p, or p considers the message lost. If p receives $estimate_{c_p}$, p adopts this new estimate, updates its local timestamp, sends an *ack* message to c_p, and, if p is not the coordinator (notice that the current coordinator also executes phase 3 but proceeds immediately to phase 4), p waits for an eventual message from the coordinator before moving to the next round. If p does not receive the *newestimate* message, p sends a *nack* message to c_p and proceeds to the next round.

4. *Phase 4* (line 28). The coordinator p ($p = c_p$) waits for messages from a majority of processes from phase 3. For each process q of a majority of processes, either p receives q's message or p considers the message lost. If p receives a majority of *ack* messages, then p knows that a majority

of processes changed their estimate to $estimate_p$, and thus $estimate_p$ is locked. The coordinator, p, sends $estimate_p$ as a *decide* message to all processes as p's last message ($S\text{-}finalsend$ primitive). Otherwise, p is unable to decide, sends a *nodecide* message to all processes, and proceeds to the next round.

The decision task

Whenever a process p receives a *decide* message in task *receive_decision* (line 35), p decides accordingly, relays the decision to all processes as p's last message, and terminates.

When no message is lost and no process crashes or is suspected, the algorithm clearly terminates in four communication steps, just as the original C&T algorithm.

4.4 Correctness

The proofs of the Agreement and Validity properties are similar to those of [4]. Therefore, we only discuss here the proof of the Termination property. The proof requires three preliminary lemmas for which we assume a majority of correct processes, a failure detector of class $\diamond\mathcal{S}$, and Stubborn communication channels.

Lemma 1. *If one correct process decides, then eventually every correct process also decides.*

PROOF (SKETCH): Let p be a correct process that decides (line 36). At line 37, process p reissues the decision message to all the processes as p's last message. As process p does not issue any further message, then by the Stubborn property of the communication channels, all correct processes receive the message and decide. □

Lemma 2. *There is a round r^* such that, either some correct process decides in round r^*, or no process proceeds to round $r^* + 1$.*

PROOF (SKETCH): By the eventual weak accuracy of $\diamond\mathcal{S}$, there is a time t after which some correct process p is no longer suspected by any correct process. Let r be the highest round that some process reaches at time t. Let r^* be the smallest round such that (1) $r^* > r$, (2) p is coordinator, and (3) all non-correct processes have crashed. Assume that no correct process decides at round r^*.

The coordinator p can proceed to round r^*+1 only if either (a) p receives a *nack* message in round r^*, or (b) p receives a message sent by some process from a round $r' > r^*$ (line 30). As no process suspects the coordinator p in round r^*, situation (a) is impossible. Situation (b) can only be possible if some process has already proceeded to round r^*+1.

A process q, that is not coordinator, can proceed to round r^*+1 only if either (c) q suspects the coordinator in round r^*, or (d) q receives a message sent by the coordinator p from a round $r' > r^*$ (line 26). As no process suspects the coordinator p in round r^*, then situation (c) is impossible. For situation (d) to occur, the coordinator p must have already proceeded to round r^*+1. Situations (a) and (d) are thus in contradiction, and hence no process proceeds to round r^*+1. □

Lemma 3. *No correct process blocks forever.*

Proof (Sketch): The proof is by contradiction. Let r be the smallest round in which some correct process p blocks forever in one of the **wait** statements. All correct processes reach round r, otherwise there is a round $r' < r$, in which a correct process blocks forever, in contradiction with the assumption that r is the smallest of such rounds. We distinguish two cases: (a) p is the coordinator and p blocks at line 12 or line 30, or (b) p is any participant and p blocks at line 20 or line 26.

Case (a) (line 12 or line 30): p is the coordinator. All correct processes have sent p an *estimate* at step $(r, 1)$ (resp. at step $(r, 3)$). The coordinator p trivially receives its own *estimate* (resp. *ack*). If any of the other correct processes decide, then by Lemma 1, p also decides. Otherwise, as by Lemma 2 no process proceeds to a round larger than round r^*, then all correct processes block (in step $(r, 3)$ or in a step larger than $(r, 3)$). By the Stubborn property of the communication channels and the assumption of a majority of correct processes, the coordinator p receives a majority of messages *estimate* (resp. *ack* or *nack*) from step $(r, 1)$ (resp. from step $(r, 3)$), or messages from larger steps. In all cases, p proceeds: a contradiction.

Case 2 (line 20 or line 26): If p is the coordinator (line 20), p trivially receives its own message from step $(r, 2)$ and proceeds. If p is a participant,

p blocks waiting for a message from the coordinator from step $(r, 2)$ (resp. from step $(r, 4)$). If the coordinator crashes, p eventually suspects the coordinator (by the strong completeness property of $\diamond\mathcal{S}$), then proceeds. If the coordinator decides, then by Lemma 1, p also decides. Otherwise, the coordinator eventually sends some message *newestimate/noestimate* (resp. *nodecide*) at step $(r, 2)$ (resp. step $(r, 4)$) and proceeds. As by Lemma 2, no process proceeds to a round larger than round r^*, the coordinator blocks (in step $(r, 4)$ or in a step larger than $(r, 4)$), and by the Stubborn property of the communication channels, p receives the coordinator's message from step $(r, 2)$ (resp. $(r, 4)$), or a message from a larger step. In all cases, p proceeds: a contradiction. □

Proposition 1. *The algorithm in Figure 2 satisfies the Termination property of Consensus with a majority of correct processes, any failure detector of class $\diamond\mathcal{S}$, and Stubborn communication channels.*

PROOF (SKETCH): Consider any correct process p in any round r. Process p can either block forever in one of the wait statements of round r, proceed to round $r + 1$, or decide (and terminate the algorithm). By Lemma 1, if any correct process decides, then every correct process eventually decides. By Lemma 2, if no correct process decides, then no process proceeds to a round larger than r^*. By Lemma 3, no process blocks forever. Hence, every correct process eventually decides. □

5 Related work

We have considered in the paper the asynchronous system model, augmented with the Chandra and Toueg failure detectors, in order to circumvent the Fischer-Lynch-Paterson impossibility result about solving the Consensus problem [8].

In [2], it is shown that any problem that is solvable with reliable channels, is also solvable with fair lossy channels (i.e., basic unreliable channels of Section 2.1). The idea underlying this result is that reliable channels can be simulated with fair lossy channels, and hence any algorithm based on reliable channels can be transformed to work with fair lossy channels. The algorithm of C&T, together with the simulation of reliable channels given in [2], can be used over fair lossy channels. Nevertheless, due to the reliable channel simulation, all messages need to be buffered and so, the memory M required to store messages increases in $O(r \cdot \log_2 r)$. In our algorithm, only the most recent message needs to be buffered, and so M increases in $O(\log_2 r)$. Figure 3 depicts the growth of M for the two algorithms. For instance, for a number of rounds r such that M in our algorithm is 32 bit, the C&T algorithm requires M to be greater than 137 Gbit (17 Gb).

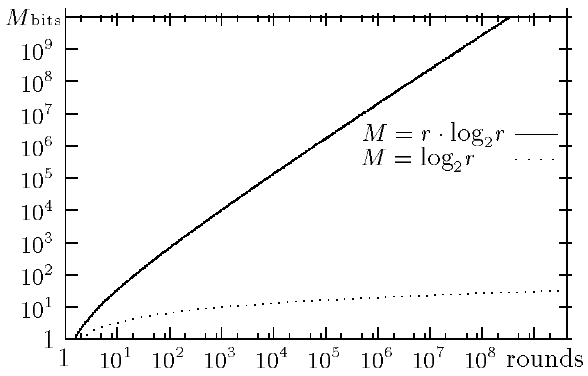

Figure 3: Storage requirements of the algorithms

The algorithm proposed in [7] also solves Consensus with basic unreliable channels. The algorithm is also a variation of the original C&T Consensus algorithm with failure detectors of class $\diamond\mathcal{S}$. The algorithm in [7] is based on an ad-hoc solution to overcome the loss of messages. We propose here a rigorous and more modular approach by introducing the notion of Stubborn channels. This notion precisely defines the delivery guarantees that enable to solve a fundamental problem like Consensus in asynchronous systems with unreliable failure detectors. Although in a different context, Rodrigues and Veríssimo proposed in [12] *transparent causal messages* in order to relax the requirements of causal communication protocols. Like messages sent through Stubborn channels, transparent messages do not prevent further messages from being sent and received at the possible cost of loosing previous messages.

In [6], Cristian and Schmuck consider an alternate model called *timed--asynchronous* model. The model distinguishes between periods during which the network is unstable (message transmission delays are unbounded, messages are lost), and periods during which the network is stable (and synchronous). In this model, Cristian and Schmuck solve the membership problem when the network is stable for a long enough period. Consensus is not considered explicitly. In comparison, our approach does not need the network to eventually become synchronous, and extends the Chandra and Toueg result by proposing a practical alternative to the reliable channel assumption. Compared to the Chandra and Toueg failure detector model, the approach in [6] explicitly handles process recovery after a crash. However, as shown in [7], process recovery can be handled by slight extensions to the Chandra and Toueg failure detector model. We believe that the failure detector approach has the advantage over the timed-asynchronous model to allow for a formal characterization of the minimal conditions under which the termination of distributed fault-tolerant algorithms is ensured.

6 Summary

Existing network layers are not fault-free: messages may be lost, reordered, and duplicated. Communication protocols, implemented on top of the network layer, are intended to mask these faults. Most of these protocols provide however the abstraction of a reliable channel only as long as no failure (link failure or process failure) occur. In case of failures, no guarantees are provided to the user, and the communication layer provides only *best-effort* semantics. Best-effort channels are not sufficient for solving Consensus-like problems in asynchronous systems with the weakest class of unreliable failure detectors ($\diamond\mathcal{S}$).

In this paper we have defined a new type of practical communication channels, called Stubborn channels. Although weakly reliable, Stubborn channels sensibly improve the reliability of existing best-effort channels, such as TCP. This improvement is biased towards the reliability requirements of fault-tolerant algorithms and we believe Stubborn channels are a useful concept to help specifying the real reliability requirements of such algorithms. In particular, we have shown that, with small changes to the Chandra and Toueg Consensus algorithm [4], the reliable channels assumption can be released, and the Consensus problem solved with Stubborn channels and $\diamond\mathcal{S}$. Furthermore, tolerating message loss does not impact on performances, as our algorithm with Stubborn channels solves Consensus with the same number of communication steps as the C&T algorithm, in situations where no process is suspected and no message is lost. This result is particularly interesting since, by tolerating general omission faults, we have shown that Consensus is solvable in a model that is a closer to practical real-world distributed systems (such as Internet).

References

[1] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, and L. Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6), July 1994.

[2] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *10th Intl. Workshop on Distributed Algorithms (WDAG'96)*. Springer Verlag, LNCS 1151, October 1996.

[3] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4), July 1996. A preliminary version appeared in *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1992.

[4] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(1), March 1996. A preliminary version appeared in *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1991.

[5] D. Comer. *Internetworking with TCP/IP: Principles, protocols, architecture*. Prentice Hall, 1988.

[6] F. Cristian and F. Schmuck. Agreeing on processor group membership in asynchronous systems. Technical Report CSE 95-428, University of California San Diego, 1995.

[7] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Cornell University, 1996.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed Consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[9] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and Consensus. In *9th Intl. Workshop on Distributed Algorithms (WDAG'95)*. Springer Verlag, LNCS 972, September 1995.

[10] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing Systems (FTCS-26)*, Sendai, June 1996.

[11] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[12] L. Rodrigues and P. Veríssimo. How to avoid the cost of causal communication in large-scale systems. In *Proceedings of the 6th SIGOPS European Workshop*, Dagstuhl, September 1994.