Structured Derivation of Semi-Synchronous Algorithms

Hagit Attiya[1]*, Fatemeh Borran[2], Martin Hutle[3]**, Zarko Milosevic[2], and André Schiper[2]

[1] Technion, Israel,
hagit@cs.technion.ac.il
[2] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland,
{fatemeh.borran | zarko.milosevic | andre.schiper}@epfl.ch
[3] Fraunhofer SIT, Germany,
martin.hutle@epfl.ch

Abstract. The semi-synchronous model is an important middle ground between the synchronous and the asynchronous models of distributed computing. In this model, processes can detect (timeout) when other processes fail. However, since detection is done by timing out, it incurs a cost much higher than the typical delay of messages.
The paper presents a new communication primitive, *Timely Announced Broadcast* (*TAB*), and uses it in algorithms for consensus and set consensus in the semi-synchronous model. Separate implementations of TAB, withstanding different types of failures, allow to derive algorithms for consensus and set consensus under crash and omission failures.
The time bounds obtained by our algorithms asymptotically match, or improve, the previously known bounds.

Keywords: semi-synchronous systems, timely announced broadcast, terminating reliable broadcast, set consensus.

1 Introduction

Most research in distributed computing considers two models, *synchronous* and *asynchronous*. In the synchronous model, processes take steps in rounds, and messages sent in one round are received by the next round; in the asynchronous model, processes take steps at arbitrary times, and there is no upper bound on message delay. Practical systems, however, are neither as predictable as the synchronous model nor as unpredictable as the asynchronous one. An important middle ground is the *semi-synchronous* model [4] in which there are bounds on the time processes take steps, or message are delivered, but they are only approximately known.

In the context of fault-tolerant distributed algorithms [3], it is assumed that consecutive steps by a correct process require time at least c_1 and at most c_2,

* part of this work done while visiting EPFL.
** part of this work done while at EPFL.

while messages are delivered within at most time d after they are sent. When failures are benign, i.e., stopping to take steps or omitting to send/receive messages, they can be detected by timing out the faulty process, according to these time bounds, since they stop sending messages. However, to avoid suspecting the innocents, timing out a process must take a relatively long time, which we denote $TO(d)$, and assume $TO(d) > d$, typically by a large margin (this is discussed further in Section 2).

A classical problem in fault-tolerant distributed computing is the *k-set consensus* problem [7], in which processes are required to output at most k *different* values; if a correct process outputs v then v must be the input of some process. The *consensus* problem is a special case where $k = 1$. In the synchronous model, exactly $\lfloor f/k \rfloor + 1$ rounds are required for solving k-set consensus in the presence of f crash failures, for any $f < n$ [7, 8]. This extends the bound for the well-known consensus problem (where $k = 1$), which can be solved in exactly $f + 1$ rounds (e.g., [10]).

These round-based algorithms can easily be *simulated* in the semi-synchronous model, by waiting to timeout all processes that did not send a message in a round, before proceeding to the next round. In this manner, executing an r-round synchronous algorithm takes roughly $r\,TO(d)$ time, namely, the timeout cost is paid for each round.

Perhaps surprisingly, this cost is not inherent, and it has been shown by Attiya, Dwork, Lynch and Stockmeyer [3] that the consensus problem can be solved in time $2fd + TO(d)$, that is, the timeout cost is paid only once (this algorithm is called ADLS). They also show that the timeout cost must be paid at least once, by proving a time lower bound of $(f - 1)d + TO(d)$. In follow-up work, Michailidis [17] presented an algorithm solving set consensus within time $\lfloor \frac{f}{k} \rfloor d + TO(2d)$.

Despite being simple, the structure of the ADLS algorithm is quite different from other consensus algorithms, and the way it works is considered "a mystery" (cf. [14]). This might be the reason there has been very little work in the semi-synchronous model.

Our contribution: In this paper, we present a new communication primitive, called *timely announced broadcast* (TAB), which simplifies the design of semi-synchronous algorithms for consensus and set consensus. TAB has simple implementations in different failure models, and we present two efficient ones for crash and omission failures. Combining these TAB implementations with simple consensus and set consensus algorithms lead to structured algorithms that match, or even improve, the best known time bounds for the semi-synchronous model.

Specifically, TAB provides three primitives to broadcast, announce and deliver a message. Informally, after a process q broadcasts m, all other processes first announce m and only then deliver m. In addition to common properties of broadcast primitives (like integrity and validity), it is guaranteed that if a correct process delivers m from q, then every correct process announces m from q. In this respect, TAB is similar to known primitives like *adopt-commit* (e.g., [13]) or *graded consensus* (e.g., [12]). Unlike these other primitives, however, TAB also provides timing guarantees, as it bounds the time duration between the broadcast, announcement, and delivery of the same message.

We present implementations of TAB in the presence of crash and omission failures. For crash failures, the time for both announcement and delivery is bounded by d, while for omission failures, the time for announcement is bounded by d and the time for delivery is bounded by $2d$. (The last algorithm assumes that $n > 2t$, where t is the maximum number of failures that can occur in an execution; we reserve f for the number of failures in a specific execution.)

We then show how TAB can be used in a simple flooding-style algorithm for *terminating reliable broadcast*, leading to a simple consensus algorithm with the same time bounds. A somewhat more elaborate, but still intuitive when considered in a synchronous setting, algorithm is needed for solving *set consensus* using TAB. Employing the TAB implementation for crash failures, we get a time bound of $fd + TO(2d)$ for consensus and a time bound of $\lfloor f/k \rfloor d + TO(2d)$ for set consensus. Employing the TAB implementation for omission failures, we get a time bound of $2fd + TO(4d)$ for consensus and a time bound of $2\lfloor f/k \rfloor d + TO(4d)$ for set consensus.

Prior bounds: Table 1 summarizes the time bounds of our algorithms and previous results.

A couple of papers extended [3] to omission failures. Ponzio [18] showed that when $n > 2t$, a simulation of crash failures on top of omission failures can be applied to the algorithm of [3] to derive an algorithm for omission failures requiring $4(f+1)d + TO(d)$ time. Berman and Bharali [6] present an improved consensus algorithm for omission failures, requiring $3(f+1)d + TO(d)$ time, when $n > 2t$. Both papers [18, 6] also present more complicated bounds for the case $n \leq 2t$.

Table 1. Comparison of our results with prior results (algorithms for omission failures assume $n > 2t$).

Problem		crash failures	omission failures
Consensus	Attiya et al. [3]	$2fd + TO(d)$	
	Michailidis [17]	$fd + TO(2d)$	
	Ponzio [18]		$4(f+1)d + TO(d)$
	Berman and Bharali [6]		$3(f+1)d + TO(d)$
	this paper	$fd + TO(2d)$	$2fd + TO(4d)$
k-set consensus	Michailidis [17]	$\lfloor f/k \rfloor d + TO(2d)$	
	this paper	$\lfloor f/k \rfloor d + TO(2d)$	$2\lfloor f/k \rfloor d + TO(4d)$

As for lower bounds, Herlihy, Rajsbaum and Tuttle [16] prove that in the semi-synchronous model, any k-set consensus algorithm for n processes and $n-1$ crash failures, requires time $\lfloor \frac{n-1}{k} \rfloor d + TO(d)$. Herlihy and Rajsbaum [15] extend this result to hold also with adversaries that fail processes in a coordinated manner. For the threshold adversary considered in our paper, where process failures are independent, they show a bound of $\lfloor \frac{f-1}{k} \rfloor d + TO(d)$, extending the previous result. These lower bounds show that our upper bounds are asymptotically optimal.

2 Preliminaries

Model of Computation: We use the model defined in [3], in which there are n processes $p_1, \ldots, p_n$, and their respective message buffers, buff_1, ..., buff_n. Each process p_i is modeled as a (possibly infinite) state machine with a local state set Q_i, including a distinguished *initial state*.

A *configuration* is a vector $s = ((q_1, b_1) \ldots, (q_n, b_n))$ where $\mathit{state}_i(s) = q_i$ is the local state of p_i and $\mathit{buff}_i(s) = b_i$ is the content of p_i's buffer. In the *initial configuration* all processes are in their initial states and all buffers are empty.

Processes communicate by sending *messages*. We assume that messages sent from p_i to p_j contain a sequence number and that the sender's id is part of every message. The action $\mathit{send}(p_j, m)$ represents the sending of message m to process p_j.

Each process p_i follows a deterministic algorithm that governs its state transitions and the messages it sends. The possible events are either *computation events* of the form $\mathit{comp}(p_i, S)$, where p_i is a process and S is a set of send actions, or *delivery events* of the form $\mathit{deliv}(p_i, m)$, where p_i is a process and m is a message.

An *execution* is an infinite sequence of alternating configurations and events $\alpha = C_0, \pi_1, C_1, \ldots, \pi_r, C_r, \ldots$, satisfying the following conditions:

1. C_0 is the initial configuration.
2. If π_r is an event of process p_i, then $\mathit{state}_j(C_{r-1}) = \mathit{state}_j(C_r)$ and $\mathit{buff}_j(C_{r-1}) = \mathit{buff}_j(C_r)$ for every $j \neq i$. That is, states and buffers of processes other than p_i do not change.
3. If $\pi_r = \mathit{comp}(p_i, S)$, then $\mathit{state}_i(C_r)$ and S are obtained by applying γ_i to $\mathit{state}_i(C_{r-1})$ and $\mathit{buff}_i(C_{r-1})$; furthermore, $\mathit{buff}_i(C_r) = \emptyset$. That is, p_i, based on its local state and the contents of its buffer, performs the send actions in S, clears its buffer and possibly changes its local state, all in one atomic transition.
4. If $\pi_r = \mathit{deliv}(p_i, m)$, then $\mathit{state}_i(C_r) = \mathit{state}_i(C_{r-1})$ and $\mathit{buff}_i(C_r) = \mathit{buff}_i(C_{r-1}) \cdot \{m\}$. That is, the message m is appended to p_i's buffer.
5. For every delivery event $\pi_r = \mathit{deliv}(p_i, m)$ there is exactly one computation event $\pi_l = \mathit{comp}(p_j, S)$ where $l < r$ and $\mathit{send}(p_i, m) \in S$. That is, each delivery is matched to a unique earlier send.

Below, 'time' is always a nonnegative real number. A *timed event* is a pair (π, t), where π is an event and t is a time. A *timed execution* is an infinite sequence of alternating configurations and timed events $\alpha = C_0, (\pi_1, t_1), C_1, \ldots, (\pi_r, t_r), C_r, \ldots$, where $C_0, \pi_1, C_1, \ldots, \pi_r, C_r, \ldots$ is an execution and the times are nondecreasing and unbounded.

Types of Failures: Fix real numbers c_1, c_2, and d, $0 < c_1 \leq c_2 < \infty$ and $0 < d < \infty$. A process p_i is *correct* in a timed execution α if the following conditions hold:

1. There is a computation event $comp(p_i, S)$ at time 0.
2. If the lth and rth timed events, $l < r$, are both computation events of p_i with no intervening computation events of p_i, then $c_1 \leq t_r - t_l \leq c_2$.
3. If a message m is sent by p_i to p_j at the lth timed event then there exists $r > l$ such that the rth timed event is the matching delivery $deliv(p_i, m)$, and $t_r - t_l \leq d$.

If a process is not correct, we say it is *faulty*, and denote by t the largest number of faulty processes that the protocol has to tolerate.

We model failure types by restricting the behavior of a faulty process. We only consider *benign* failures, where faulty processes follow the algorithm. With *crash failures* a faulty process may stop taking steps (or not start at all), but its messages are delivered on time. Specifically, every message that is sent at time T is received the latest at time $T + d$, if the receiver process is correct.

Omission failures are slightly more severe than crash failures, and although they ensure the delivery times of messages, messages sent by a faulty process or to faulty process may not be delivered at all. Specifically, every message that is received at time T is sent not before time $T - d$. Every message that is sent by a correct process is received if the receiver is correct.

Following the literature on *early-stopping* consensus and set consensus, we denote by f the number of processes that fail in a specific execution of the algorithm, assuming that the execution is clear from the context. We reserve t to denote the maximum number of failures that is possible in all executions.

A Timeout Task: Let $TO(T)$ be the worst-case time to detect that time T has elapsed. In order to ensure that time T has elapsed, a process must count $\frac{T}{c_1}$ of its own steps,[4] since it might be running "fast" (i.e., time c_1 between steps). But if the process is actually running "slow" (i.e., time c_2 between steps), the actual waiting time is $\frac{c_2 T}{c_1}$. That is, $TO(T) = CT$, with $C = \frac{c_2}{c_1}$.

In order to detect the failure of process p, processes must ensure that no messages from p are in transit. Therefore, the worst-case elapsed time between the failure of p and the time when all correct processes determine that p has failed is roughly Cd.

3 Timely Announced Broadcast

We introduce a new communication primitive, *Timely Announced Broadcast (TAB)*, and show how it can be used to solve the consensus and set consensus problems. TAB is defined in terms of three primitives, *ta-broadcast*(m),

[4] We ignore rounding issues and assume that c_1 always divides T and c_2.

Algorithm 1 TAB with crash failures; code for process p.

```
upon ta-broadcast($m$) do
  send ⟨ANNOUNCE, $m, p$⟩ to all
  send ⟨MSG, $m, p$⟩ to all

upon received ⟨ANNOUNCE, $m, q$⟩ do
  if not announced $m$ from $q$ yet then
    announce($m, q$)

upon received ⟨MSG, $m, q$⟩ do
  if not announced $m$ from $q$ yet then
    announce($m, q$)
  ta-deliver($m, q$)
```

announce(m, q), and *ta-deliver*(m, q). Informally, after a correct process q invokes *ta-broadcast*(m), all other processes first *announce*(m, q) and only then *ta-deliver*(m, q).

The announce message indicates to a process to wait for a forthcoming message, causing it to extend its timeout and wait enough time to deliver the expected message (as demonstrated in Section 4).

In addition to common properties of broadcast primitives (like integrity and validity), it is guaranteed that if a correct process ta-delivers m from p, then every correct process announces m from p. TAB also provides timing guarantees, as it bounds the time duration between the broadcast, announcement, and delivery of the same message. In our implementations of TAB, these bounds are in $O(d)$, with the constant being small, i.e., 1 or 2.

Definition 1 (TAB). *An algorithm solves* timely announced broadcast in the presence of benign failures*, with two parameters $d_1 \geq d_2 > 0$, if the following properties hold:*

Integrity *If a process ta-delivers a message m from p, then m was ta-broadcast by p.*

Validity *If a correct process p ta-broadcasts a message m at time T, then all correct processes eventually announce m from p and ta-deliver m from p the latest at time $T + d_1$.*

Announcement *For any message m, if any process ta-delivers m from p at time T, then every correct process announces m from p the latest at time $T + d_2$.*

Implementing TAB in the presence of crash failures: Algorithm 1 implements TAB, with crash failures and assuming $n > t$. It is easy to verify that the algorithm satisfies the properties of Definition 1, with $d_1 = d_2 = d$.

Implementing TAB in the presence of omission failures: Algorithm 2 implements TAB, with omission failures and assuming $n > 2t$. It is simple to show that the

Algorithm 2 TAB with omission failures and $n > 2t$; code for process p.

```
upon ta-broadcast($m$) do
    send $\langle \textsc{Msg}, m \rangle$ to all

upon received $\langle \textsc{Msg}, m \rangle$ from $q$ do
    send $\langle \textsc{Ack}, m, q \rangle$ to all

upon received $\langle \textsc{Ack}, m, q \rangle$ the first time do
    announce($m, q$)

upon received $t + 1$ $\langle \textsc{Ack}, m, q \rangle$ do
    ta-deliver($m, q$)
```

algorithm satisfies the properties of Definition 1, with $d_1 = 2d$ and $d_2 = d$. Inspecting the code verifies that the integrity property holds.

Lemma 1 (Integrity). *If a process ta-delivers m from p, then m was ta-broadcast by p.*

Lemma 2 (Validity). *If a correct process p ta-broadcasts a message m at time T, then all correct processes eventually announce m from p and ta-deliver m from p the latest at time $T + d_1$, where $d_1 = 2d$.*

Proof. Since p is correct and ta-broadcasts m at time T, it sends $\langle \textsc{Msg}, m \rangle$ to all by time T, and by time $T + d$, all correct processes send $\langle \textsc{Ack}, m, p \rangle$ to all.[5] Since $n > 2t$, this means that by time $T + 2d$, every process receives at least $n - t \geq t + 1$ $\langle \textsc{Ack}, m, p \rangle$ messages and therefore announces and ta-delivers m from p. □

Lemma 3 (Announcement). *For any message m, if any process ta-delivers m from p at time T, then every correct process announces m from p the latest at time $T + d_2$, where $d_2 = d$.*

Proof. If a process delivers m from p at time T, then it received $t+1$ $\langle \textsc{Ack}, m, p \rangle$ messages by time T, at least one of them was sent by a correct process q by time T. Then by time $T + d$, every correct process receives an $\langle \textsc{Ack}, m, p \rangle$ message from q and announces m from p. □

4 Terminating Reliable Broadcast from TAB

The *Terminating Reliable Broadcast (TRB)* problem is defined in terms of two primitives, broadcast and deliver, and has a dedicated sending process s. The sender process s is the only process that invokes *broadcast*. When failures are benign, we have the following requirements.

[5] To simplify the statements of the results and the proofs, we assume $c_2 \ll d$ and approximate $d + c_2$ with d.

Definition 2 (TRB). *An algorithm solves* terminating reliable broadcast in the presence of benign failures *if the following properties hold:*

Integrity *A process delivers at most one message, and if a process delivers a message $m \neq \perp$, then m was broadcast by s.*
Validity *If the sender s is correct and broadcasts a message m, then s eventually delivers m.*
Agreement *If a correct process delivers a message m, then every correct process delivers m.*
Termination *Every correct process eventually delivers some message.*

Consensus from TRB: Non-uniform consensus can be easily implemented from TRB with a simulation, where n instances of the TRB algorithm are executed in parallel, in each instance i process i is the sender and uses its initial value for that, and all processes apply a deterministic function on the resulting vector. The response time of this consensus algorithm equals the response time of the TRB algorithm.

TRB from TAB: It is easy to solve TRB in a synchronous system, using a familiar, simple *flooding* mechanism. In the simplest form of this protocol (cf. [5, Algorithm 15]), for a synchronous system with crash failures, the sender sends a message to all processes; each process that gets this message, echoes it by re-sending it to all processes, and returns the value. If no value was returned after $f + 1$ rounds, the process returns $\perp$.

This algorithm can be deployed in a semi-synchronous system by timing out the processes that failed at the beginning of the round before correct processes advance to the next round. This algorithm however, is susceptible to timing delays, since timing out a process takes $TO(d)$ and hence timing out r rounds may take $r\,TO(d)$ time, yielding a $(f + 1)Cd$-time algorithm for consensus.

To overcome this problem, instead of sending and receiving messages directly, processes use TAB to send and announce-deliver messages. The TAB protocol allows processes to warn other processes that they are about to deliver a message, thus alerting them to wait for their copy of this message.

The pseudocode is given as Algorithm 3. For a process p, Z_p holds the set of processes who have sent an announcement and p must wait for their message. For any process $q \in Z_p$, if process p does not receive a message from q within a specific time, namely $2d_1$, it suspects q to be faulty and stops waiting for a message from q by removing q from Z_p (line 11). When $Z_p = \perp$, all processes who have only sent an announcement are crashed, so it is safe to deliver $\perp$ (lines 12-13). We show the algorithm solves TRB.

Lemma 4 (Integrity). *A process delivers at most one message, and if a process delivers a message $m \neq \perp$, then m was broadcast by s.*

Proof. The first part of the lemma follows from the code (a process stops after delivering a message). Assume that a process p delivers $m \neq \perp$. Since a non-$\perp$ message is delivered only after a *ta-deliver* event at line 9, this means that

Algorithm 3 TRB from TAB; code for process p (s is the sender process)

```
1:  initially
2:      $Z_p \leftarrow \{s\}$
3:  upon broadcast $v$ do                                  /* called only by $s$ */
4:      $\textit{ta-broadcast}(v)$
5:  upon $\textit{announce}(m, q)$ do      /* a message from $q$ is forthcoming, wait for it */
6:      $Z_p \leftarrow Z_p \cup \{q\}$
7:  upon $\textit{ta-deliver}(v, q)$ the first time
    do                    /* a message from $q$ ta-delivered, echo it and deliver */
8:      $\textit{ta-broadcast}(v)$
9:      deliver $v$
10: upon no $\textit{ta-deliver}(v, q)$ for $2d_1$ time since the last $\textit{announce}(v, q)$ or time 0
    do                                                            /* suspect $q$ */
11:     $Z_p \leftarrow Z_p \setminus \{q\}$              /* stop waiting for all messages from $q$ */
12:     if $Z_p = \emptyset$ then
13:         deliver $\perp$
```

p ta-deliver (m, q). By the Integrity property of TAB, m was ta-broadcast by process q. If $q = s$, this finishes the proof. Otherwise, q ta-broadcast value it ta-deliver from some process. Either q ta-deliver message from s or after a chain of processes, where by the Integrity property of TAB and the fact that no process executes line 8, at least one of processes in the chain will ta-deliver message from process s. Therefore, if p deliver a non-$\perp$ message, then it must be m. □

Lemma 5 (Validity). *If the sender s is correct and broadcasts a message m, then s eventually delivers m.*

Proof. From the Validity property of TAB, s ta-delivers m the latest at time d_1. Since at this time s cannot execute the upon rule of Line 10, s delivers m by Line 9. □

Lemma 6 (Agreement). *If a correct process delivers a message m, then every correct process delivers m.*

Proof. By contradiction. Because of Lemma 4, w.l.o.g. assume that at time T_p, a correct process p delivers v, the value broadcast by s, and at time T_q, a correct process $q \neq p$ delivers $\perp$.

If process q delivers $\perp$ at time T_q, it did not ta-deliver nor receive an announcement for $2d_1$ time (a ta-delivery would have led to deliver v and an announcement would have delayed the delivery of $\perp$). Thus, process p cannot deliver before time $T_q - d_1$: before delivering it would have ta-broadcast its message to all, and thus, since p is correct, q would have received this message. Therefore, $T_p > T_q - d_1$. Because p delivered v, it ta-delivered a message; in more detail, there is a chain of deliveries from the source to p. Each of these ta-broadcast events is at most d_1 apart from each other. Since $T_q \geq d_1$, and the source ta-broadcasts at time 0, there is one process in this chain that ta-broadcasts after $T_q - 2d_1$ but before $T_q - d_1$. Because of the Announcement property, an announcement is received by q between $T_q - 2d_1$ and $T_q - d_1 + d_2 \leq T_q$. This contradicts the fact, that q neither ta-delivers nor receives an announcement between $T_q - 2d_1$ and T_q. □

The upon rule of Line 10 ensures the next lemma.

Lemma 7 (Termination). *Every correct process eventually delivers some message.*

The next lemma shows the timing property of the TRB algorithm.

Lemma 8. *In a run with an actual TAB transmission time d_1 and f faulty processes that obey the timing requirements, a correct process delivers a value by time $fd_1 + TO(2d_1)$.*

Proof. By Lemma 7, a correct process p eventually delivers a value for the sender s. If p delivers $m \neq \perp$ in Line 9, then it can be easily shown (cf. [5, Algorithm 15]) that it has received m along a chain of r re-broadcasts by different processes. It follows that $r \leq f + 1$, since once m reaches a correct process it is sent to all processes. Thus, it is delivered by time $(f+1)d_1 = fd_1 + d_1 \leq fd_1 + TO(d_1)$ (since $TO(d) \geq d$).

Consider now the case p delivers $\perp$ in Line 13; we argue that each process q added to Z_p is removed by time $fd_1 + TO(2d_1)$, and the upon rule of Line 10 ensures $\perp$ is delivered.

A process q is added to Z_p in the upon rule of Line 5. Since no process announces the same process twice (by Line 7), an announcement is forwarded through a chain of r different processes $p_{i_1}, \ldots, p_{i_r}$.

If none of these processes is correct, then $r \leq f$, and hence q is added in the upon rule of Line 5 before time fd_1, and it is timed out (in the upon rule of Line 10) before time $fd_1 + TO(2d_1)$, implying the claim.

So, let $p_{r'}$ be the first correct process among $p_{i_1}, \ldots, p_{i_r}$; clearly, $r' \leq f + 1$, and hence p receives the ta-broadcasts from $p_{r'}$ before time $r'd_1 \leq (f+1)d_1 \leq fd_1 + TO(2d_1)$ (since $TO(d) \geq d$). □

By substituting the appropriate TAB implementations, we get:

Corollary 1. *There is a TRB algorithm, and hence consensus algorithm, which withstands crash failures and terminates within time $fd + TO(2d)$.*

Corollary 2. *There is a TRB algorithm, and hence consensus algorithm, which withstands omission failures and terminates within $2fd + TO(4d)$, assuming that $n > 2t$.*

Algorithm 4 Set Consensus from TAB; code for process p.

```
1:  initially
2:     $\forall q \in \Pi : know_p[q] \leftarrow \bot$
                    /* $know_p[q]$ contains the initial value of process $q$ learned by $p$ */
3:     $\forall q \in \Pi : Z_p[q] \leftarrow \{q\}$
          /* set of processes that announced the knowledge of the initial value of $q$ */

4:  upon starting with input $v$ do
5:     $know_p[p] \leftarrow v$
6:     $ta\text{-}broadcast(know_p)$

7:  upon $announce(kn, q)$ do
8:     for all $r : kn[r] \neq \bot$ do
9:        $Z_p[r] \leftarrow Z_p[r] \cup \{q\}$
                    /* $q$ learned $r$'s initial value but $p$ didn't, wait for this value */

10: upon $ta\text{-}deliver(kn, q)$ do
11:    for all $r \in \Pi$ do
12:       if $know_p[r] = \bot$ and $kn[r] \neq \bot$ then
13:          $know_p[r] \leftarrow kn[r]$                /* learn initial values $q$ knows */
14:    if updated $know_p$ then
15:       $ta\text{-}broadcast(know_p)$     /* inform others about new learned initial values */
16:    if can-decide? then
17:       decide $\min(know_p)$

18: upon no $ta\text{-}deliver(v, q)$ for $2d_1$ time since the last $announce(v, q)$ or time 0
     do                                                             /* suspect $q$ */
19:    for all $r \in \Pi$ do
20:       $Z_p[r] \leftarrow Z_p[r] \setminus \{q\}$              /* stop waiting for all messages from $q$ */
21:    if can-decide? then
22:       decide $\min(know_p)$

23: function can-decide?
24:    $K_p \leftarrow \{q : know_p[q] = \bot\}$ /* set of processes that $p$ doesn't know their value */
25:    $U_p \leftarrow \{q : \exists r \text{ s.t. } know_p[r] = \bot \ \wedge \ q \in Z_p[r]\}$
        /* set of processes that know values of those processes that $p$ doesn't know */
26:    return $(|K_p| < k)$ or $(|U_p| < k)$
```

5 Set Consensus from TAB

Definition 3 (Set consensus). *An algorithm solves k-*set consensus *in the* presence of benign failures *if each process starts with an input value, and decides on a value, such that the following properties hold:*

Integrity *If a process decides v, then v is the input of some process.*
Agreement *The correct processes decide on at most k different values.*
Termination *Every correct process eventually decides.*

The pseudocode for set consensus appears in Algorithm 4. Recall that all processes start at time 0. Each process p keeps an array of known initial values;

$know_p[q]$ is the initial value of process q learned by p, and is initially $\perp$. During the algorithm, process p *learns* the initial values of other processes. Unlike TRB that has a unique source, in set consensus all processes are sources. Therefore, process p keeps a set Z_p for each process q: $Z_p[q]$ denotes the set of processes who sent an announcement for q, that is, $Z_p[q]$ is the set of processes who have learned the initial value of process q. As for Algorithm 3, if process p does not receive a message from q within the specified time, namely $2d_1$, it suspects q to be faulty and stops waiting for a message from q (lines 18-20).

The decision condition is also more complicated that in TRB. As in synchronous set consensus algorithms [7], process p can decide if it knows more than $n-k$ values, i.e., $|K_p| < k$, where K_p denotes the set of processes that p does not know their initial value. Additionally, process p can decide if fewer than k processes know the initial value of those processes that p does not know, i.e., the size of $U_p \triangleq \{q : \exists r \text{ s.t. } know_p[r] = \perp \ \wedge \ q \in Z_p[r]\}$ is strictly less than k. That is, if only $k-1$ processes know some values that p does not know, at most $k-1$ different values might be decided; therefore, p can decide on the minimum value that it knows.

The correctness proof follows arguments similar to those used to prove the correctness of Algorithm 3. A process decides only on a value from the *know* array; the values in this array are either the process's initial value or those received in a *ta-deliver* event. Hence, the Integrity property of TAB implies the next lemma.

Lemma 9 (Integrity). *If a process decides v, then v was proposed by some process.*

Lemma 10 (Agreement). *The correct processes decide on at most k different values.*

Proof. Assume, towards a contradiction, that at least $k+1$ different values, $v_1 < \ldots < v_{k+1} < \ldots$, are decided. Let p be a correct process that decides v_{k+1}. By Line 26, p decides either because $|K_p| < k$ or because $|U_p| < k$.

If $|K_p| < k$, then since p decides on the minimum value it has seen it follows that it has not seen $v_1, \ldots, v_k$, that is, k values, which is a contradiction.

Otherwise, p decides because $|U_p| < k$. Note that this happens only when the condition of Line 18 is satisfied, i.e., $2d_1$ timeout expires. $|U_p| < k$ implies that (i) at most $k-1$ processes know values that p doesn't know. This means that at most k different values are decided (including p's decision value). By the assumption, (ii) there are at least $k+1$ decisions.

From (i) and (ii), it follows that there is a decision value $x < v_{k+1}$, such that every process in U_p that knows x also knows a value smaller than x. Let q be a correct process that decides x. We consider two cases for how q has received x:

Case 1, q receives x after p decides: Since p decided v_{k+1} and $x < v_{k+1}$, p does not know x, this implies that q did not receive x from p. Therefore, q must have received x from some process in U_p, possibly through other intermediate processes. But every process in U_p that knows x also knows a value smaller than x, a contradiction.

Case 2, q receives x before p decides: q must appear in $Z_p[r]$, for some $r \in \Pi$, since q ta-broadcasts x before deciding. Since p does not know x and p has timed out all faulty processes by Line 18, p must have detected q's failure, a contradiction. □

The upon rule of Line 18 together with the properties of TAB ensure the next lemma.

Lemma 11 (Termination). *Every correct process eventually decides.*

Proof. We show that every correct process continues to take steps until it decides or crashes, i.e., the condition of Line 26 eventually becomes true.

Assume, by way of contradiction, that some correct process p continues to take steps without deciding. This means that the sizes of the sets K_p and U_p it has are greater than or equal to k (according to Line 26). Thus, there are at least k unknown values and at least k processes in $Z \triangleq \bigcup_{r \in \Pi} Z_p[r]$. Consider one of these processes, say q. Since $q \in Z$, it has ta-broadcast $know_q$. If q is a correct process, by the Validity property of TAB, p ta-delivers a message from q within d_1. Otherwise, p suspects q after $2d_1$ according to Line 18. In either case, the condition of Line 26 becomes satisfied, which contradicts the fact that p never decides. □

The final lemma shows the timing property of the set consensus algorithm.

Lemma 12. *In a run with an actual TAB transmission time d_1 and f faulty processes that obey the timing requirements, a correct process decides by time $\lfloor f/k \rfloor d_1 + TO(2d_1)$.*

Proof. (Sketch) By Lemma 11, a correct process p eventually decides. Let p decide v, where v is the initial value of some process q, i.e., $know_p[q] = v$. From Line 26, p decides because either (i) $|K_p| < k$ or (ii) $|U_p| < k$.

In case (i), it can be easily shown (cf. [7]) that p has received v along a chain of $r \leq \lfloor f/k \rfloor + 1$ re-broadcasts by different processes. Thus v is decided by time $(\lfloor f/k \rfloor + 1)d_1 \leq \lfloor f/k \rfloor d_1 + TO(d_1)$ (since $TO(d) \geq d$).

In case (ii) we show that each process $q' \in Z_p[q]$ is removed by time $\lfloor f/k \rfloor d_1 + TO(2d_1)$. Process q' receives v through a chain of r different processes.

If none of these processes is correct, then $r \leq \lfloor f/k \rfloor$. Therefore, q' is added to $Z_p[q]$ by time $rd_1 \leq \lfloor f/k \rfloor d_1$ in the upon rule of Line 7 and is removed from $Z_p[q]$, after a timeout $TO(2d_1)$, by time $rd_1 + TO(2d_1) \leq \lfloor f/k \rfloor d_1 + TO(2d_1)$ in the upon rule of Line 18, which implies the lemma.

Otherwise, p receives v from a correct process by time $r'd_1$, where $r' \leq \lfloor f/k \rfloor + 1$. Therefore, p decides v by time $(\lfloor f/k \rfloor + 1)d_1 \leq \lfloor f/k \rfloor d_1 + TO(2d_1)$ (since $TO(d) \geq d$). □

By substituting the appropriate TAB implementations, we get:

Corollary 3. *There is a k-set consensus algorithm, which withstands crash failures and terminates within time $\lfloor f/k \rfloor d + TO(2d)$.*

Corollary 4. *There is a k-set consensus algorithm, which withstands omission failures and terminates within $2\lfloor f/k \rfloor d + TO(4d)$, assuming that $n > 2t$.*

6 Summary

This paper presents a new communication primitive and uses it to derive consensus and set consensus algorithms for semi-synchronous systems, under several types of failures. The time bounds achieved by our algorithms asymptotically match or improve previously known bounds, but we consider the main contribution of our paper to be the modular structure of our algorithms, which provides insight into the behavior of efficient semi-synchronous algorithms.

The time bounds of our algorithms are the sum of two terms: one depending only on d and another depending on a timeout (which itself depends on d). Interestingly, it can be shown that the first term is even smaller in some executions. Let δ be the maximum transmission delay *of a certain execution.* Then the execution time of, e.g., our consensus algorithm for crash failures is in fact $f\delta + TO(2d)$, which is important for the case $\delta \ll d$. (The other bounds can be adjusted similarly.)

We remark that our algorithms are *early stopping*, since their time bounds depend on f, the actual number of failures in an execution, rather than on t, the maximal number of failures. Thus, overall, the execution time of these algorithms is a constant (in terms of f) plus a term that depends only on the actual properties of an execution, f and δ (and not t and d).

The most obvious open question is to tighten the time bounds, especially for omission failures.

It is also interesting to study how TAB (or some extension thereof in the style of [9]) can yield algorithms that withstand timing or Byzantine failures. Ponzio [18] showed that in the presence of Byzantine failures, consensus can be solved in $(f+1)(d+TO(d))$. Attiya and Djerassi-Shintel [2] prove lower bounds in the presence of t *timing* failures. Specifically, they showed any consensus algorithm requires $\Omega(\frac{n}{n-t}TO(d))$ time, while a k-set consensus algorithm requires $\Omega(\frac{n}{k(n-t)}TO(d))$ time. This leaves a gap for small values of t.

Taking a broader perspective, can TAB be used to derive efficient algorithms for other problems, or even as a general technique for simulating synchronous algorithms?

The *partially synchronous* model [11] considers an asynchronous system and requires algorithms to terminate only after the system experiences a long enough synchronous period. (This is also known as the *eventually synchronous* model.) It would be intriguing to investigate implementing TAB in this model, and using it to efficiently solve problems such as consensus and set consensus. A key step would be to make our algorithms work even when not all process start at the same time (non-synchronized start).

Aguilera, Le Lann and Toueg [1] show how fast failure detection can speed up consensus in a synchronous system; their results are similar to [3]. However, as explained in their paper, “specialized hardware” or “different messaging service” are required to achieve fast failure detection. This is in contrast to the ADLS model, studied in our paper, which assumes that all messages sent in this model take at most time d.

References

1. Marcos Kawazoe Aguilera, Gérard Le Lann, and Sam Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *DISC*, pages 354–370, 2002.
2. Hagit Attiya and Taly Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *Journal of Parallel and Distributed Computing*, 61(8):1096–1109, 2001.
3. Hagit Attiya, Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.
4. Hagit Attiya and Nancy A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Information and Computation*, 110(1):183–232, 1994.
5. Hagit Attiya and Jennifer L. Welch. *Distributed computing: Fundamentals, simulations, and advanced topics*. Wiley-Interscience, 2004.
6. Piotr Berman and Anupam A. Bharali. Distributed consensus in semi-synchronous systems. In *IPPS*, pages 632–635, 1992.
7. S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 103(1):132–158, 1993.
8. Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912–943, 2000.
9. Brian A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, December 1988.
10. Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.
11. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
12. Paul Feldman and Silvio Micali. Optimal algorithms for Byzantine agreement. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 148–161, 1988.
13. Eli Gafni. Round-by-round fault detectors: unifying synchrony and asynchrony. In *Proceedings of the 18th annual ACM symposium on Principles of distributed computing (PODC '98)*, 1998.
14. Maurice Herlihy. Public communcation. http://www.youtube.com/watch?v=s6uEsO2T2lg, minutes 9:28–9:37.
15. Maurice Herlihy and Sergio Rajsbaum. Concurrent computing and shellable complexes. In *Proceedings of the 24th international conference on Distributed computing (DISC)*, pages 109–123, 2010.
16. Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the 18th annual ACM symposium on Principles of distributed computing (PODC '98)*, pages 133–142, 1998.
17. Dimitris Michailidis. Fast set agreement in the presence of timing uncertainty. In *Proceedings of the 18th annual ACM symposium on Principles of distributed computing (PODC '99)*, pages 249–256, 1999.
18. Stephen Ponzio. Consensus in the presence of timing uncertainty: omission and byzantine failures. In *Proceedings of the 10th annual ACM symposium on Principles of distributed computing (PODC '91)*, pages 125–138, 1991.