Fault-Tolerant Consensus in Unknown and Anonymous Networks

Carole Delporte-Gallet, Hugues Fauconnier and Andreas Tielmann
LIAFA, University Paris VII, France
{cd, hf, tielmann}@liafa.jussieu.fr

Abstract

This paper investigates under which conditions information can be reliably shared and consensus can be solved in unknown and anonymous message-passing networks that suffer from crash-failures. We provide algorithms to emulate registers and solve consensus under different synchrony assumptions. For this, we introduce a novel pseudo leader-election approach which allows a leader-based consensus implementation without breaking symmetry.

1. Introduction

Most of the algorithms for distributed systems consider that the number of processes in the system is known and every process has a distinct ID. However, in some networks such as in wireless sensors networks, this is not necessarily true. Additionally, such networks are typically not totally synchronous and processes may suffer from failures such as crashes.

Designing protocols for such networks is especially intricate, since a process can never know if its messages have been received by all processes in the system. In this paper, we investigate under which conditions information can be reliably shared and consensus can be solved in such environments.

Typically, in systems where no hardware registers are available, one makes additional assumptions to be able to reliably share information, e.g. by assuming a correct majority of processes. However, these techniques assume also some knowledge about the total number of processes. With processes with distinct identities, the requirements to emulate a register have been precisely determined by showing that the quorum failure detector $\Sigma$ is the weakest failure detector to simulate registers in asynchronous message passing systems [5]. But again, this approach fails due to the lack of identities in our anonymous environment.

To circumvent these problems, we assume that the system is not totally synchronous, but assume the existence of some partial synchrony. We specify our environments by using the general round-based algorithm framework (GIRAF) of [12]. This has two advantages: (i) it is easy to precisely specify an environment and (ii) it makes it easy to emulate environments to show minimality results.

We first define the moving source environment (MS) in which at every time at least one process (called the source) sends timely messages to all other processes, but this source may change over time and infinitely often. Although this environment is considerably weaker than a total synchronous environment, we show that it is still sufficient to implement registers, although it is not possible to implement the consensus abstraction. In fact, it can be emulated by hardware registers in totally asynchronous “known” networks for any number of process crashes. Therefore, if we would be able to implement consensus in this environment, we could contradict the famous FLP impossibility result [8].

This result states, that consensus cannot be implemented in asynchronous message passing networks, even if only one process may crash. Since we can emulate registers if only one process may crash [2], we can also emulate the MS environment and therefore cannot be more powerful.

To implement consensus, we consider some additional stronger synchrony assumptions. Our first consensus algorithm assumes that additionally to the assumptions of the MS environment, eventually all processes communicate timely. We call this environment the eventual synchronous (ES) environment. It resembles Dwork et al. [7]. In our second consensus algorithm, we consider a weaker environment and only assume that eventually always the same process is able to send timely to all other processes. We call it the eventual stable source environment (ESS). It resembles the model of [1] in which it is used to elect a leader, a classical approach to implement in turn consensus.

Due to the indistinguishability of several processes that behave identical, a true leader election is not possible in our anonymous environment. Therefore, in our second algorithm, we take benefit of the fact that it suffices for the implementation of consensus if all processes that consider itself as a leader behave the same way. We show how to eventually guarantee this using the history of the processes proposal values.

Furthermore, we consider the weak-set data-structure [4]. This data-structure comes along some problems that arise with registers in unknown and anonymous networks. Every process can add values to a weak-set and read the values written before. Contrary to a register, it allows for sharing information without knowing identities of other processes.

Carole Delporte-Gallet and Hugues Fauconnier were supported by grant ANR-08-VERSO-SHAMAN. Andreas Tielmann was supported by grants from Région Ile-de-France.
and without the risk of an overwritten value due to a concurrent write. Furthermore, we show that it precisely captures the power of the MS environment, i.e. we can show that it can be implemented in the MS environment and a weak-set can be used to emulate the MS environment. Interestingly, in known networks, a weak-set is equivalent to the register abstraction and can thus be seen as a generalization for unknown and anonymous networks.

Furthermore, we show that although it is possible to emulate registers in our MS environment, it is not possible to emulate $\Sigma$ [5], the weakest failure detector for registers. And this result is not only due to the anonymity of the processes, it holds even if the number of processes and their identities are known. Note that this is not a contradiction, since the result in [5] means only that $\Sigma$ is the weakest of all failure detectors with which a register can be implemented and we have exhibited synchrony assumptions where the existence of a failure detector is not necessary at all.

1. Related work

There have been several approaches to solve fault-tolerant consensus in anonymous networks deterministically. In [4], fault-tolerant consensus is solved under the assumption that failure detector $\Omega$ [3] exists, i.e. exactly one correct process eventually knows forever that it is the leader. In [10], fault-tolerant and obstruction-free$^1$ consensus is solved if registers are available.

There has also been some research on systems where IDs are known but the number of processes is not. In [9], it is assumed that processes may crash, but furthermore that it is possible to detect the participants initially. In [13], a leader election algorithm for a system where infinitely many processes may join the system is presented if the number of processes simultaneously up is bounded.

To the best of our knowledge, this paper presents completely new approaches to emulate registers and solve the consensus problem in unknown and anonymous environments with partial synchrony.

2. Model and Definitions

We assume a network with an unknown (but finite) number of processes where the processes have no IDs (i.e. they are totally anonymous) and communicate using a broadcast primitive. The set of processes is denoted $\Pi$. We assume that the broadcast primitive is reliable, although it may not always deliver messages on time. Furthermore, any number of processes may crash and the processes do not recover. Processes that do not crash are called correct.

We model an algorithm $A$ as a set of deterministic automata, one for every process in the system. We assume only fair runs, i.e. every correct process executes infinitely many steps.

2.1. Consensus

In the consensus problem, the processes try to decide on one of some proposed values. Three properties have to be satisfied:

- **Validity:** Every decided value has to be a proposed value.
- **Termination:** Eventually, every correct process decides.
- **Agreement:** No two processes decide different values.

2.2. An extension to GIRAF

Algorithm 1 presents an extension to the generic round-based algorithm framework of [12] (GIRAF). It is extended to deal with the particularities of our model, namely the anonymity and unknown number of the processes. The framework is modeled as an I/O automaton. To implement a specific algorithm, the framework is instantiated with two functions: `initialize()` and `compute()`. The `compute()` function takes the round number and the messages received so far as parameters. We omit to specify a failure detector output as parameter (as in [12]), because we are not interested in failure detectors here. Both functions are non-blocking, i.e. they are not allowed to wait for any other event.

Our extension lies in the way we model the received messages. Since the processes have no IDs, we represent the messages that are received during one round as a set instead of an array.

The communication between the processes proceeds in rounds and the advancement of the rounds is controlled by the environment via the `receive()` and `end-of-round()` input actions. These actions may occur separately at each process $p_i$ and therefore rounds are not necessarily synchronized among processes. The framework can capture any asynchronous message passing algorithm (see [12]).

Environments are specified using round-based properties, restricting the message arrivals in each round.

2.3. Environments

We say that a process $p_i$ is in round $k$, if there have been $k$ invocations of `end-of-round()`. A process $p_i$ has a **timely link in round $k$**, if `end-of-round()` occurs in round $k$ and every correct process $p_j$ receives the round $k$ message of $p_i$ in round $k$.

In this paper, we consider three different environments:

- In the first one, which we call the moving-source (MS) environment, we assume that in every round $k$, there

$^1$ For obstruction-free consensus, termination is only guaranteed if a process can take enough steps without being interrupted by other processes.
Algorithm 1: Extended GIRAF generic algorithm for process \( p_i \).

1. States:
   1. \( k_i \in \mathbb{N} \), initially 0;
   2. \( M_i[N] \subseteq \text{Messages} \), initially \( \forall k \in \mathbb{N} : M_i[k] = \emptyset \);

2. Actions and Transitions:
   1. input end-of-round,
      1. if \( (k_{i} = 0) \) then
         1. \( m := \text{initialize}() \);
      2. else
         1. \( m := \text{compute}(k_{i}, M_i) \);
         2. \( M_i[k_{i} + 1] := M_i[k_{i} + 1] \cup \{ m \} \);
         3. \( k_{i} := k_{i} + 1 \);
   2. output send((\( M_i[k_{i}], k_{i} \)));
   3. input receive((M, k)),
      1. \( M[k] := M[k] \cup M \);

exists a process \( p_s \) (a source) that has a timely link in round \( k \).

- In the second environment, which we call the eventual synchronous (ES) environment, we demand the same as in the MS environment, but additionally require that there is some round \( k \) such that in every round \( k' \geq k \), all correct processes have timely links in round \( k' \).

- In the third environment, which we call the eventually stable (ESS) environment, we demand the same as in the MS environment, but additionally require that eventually the source process \( p_s \) is always the same in every round. This means, that there is some round \( k \) such that in every round \( k' \geq k \), the same process \( p_s \) has a timely link in round \( k' \).

3. Implementing consensus in ES

Algorithm 2: A consensus algorithm in ES for process \( p_i \).

1. on initialization do
   1. \( \text{VAL} := \text{initial value} \);
   2. \( \text{WRITTEN} := \text{WRITTENOLD} := \text{PROPOSED} := \emptyset \);
   3. return PROPOSED;
   4. on compute(\( k_{i}, M_i \)) do
      1. \( \text{WRITTEN} := \bigcup_{m \in M_i[k_{i}]} m \);
      2. \( \text{PROPOSED} := (\bigcup_{m \in M_i[k_{i}]} m) \cup \text{PROPOSED} \);
      3. if \( (k_{i} \mod 2 = 0) \) then
         1. if \( (\text{PROPOSED} = \text{WRITTENOLD} = \{ \text{VAL} \}) \) then
            1. decide VAL; halt
         2. else if \( (\text{WRITTEN} \neq \emptyset) \) then
            1. \( \text{VAL} := \max(\text{WRITTEN}) \);
            2. \( \text{PROPOSED} := \{ \text{VAL} \} \);
         3. \( \text{WRITTENOLD} := \text{WRITTEN} \);
      4. return PROPOSED;

Algorithm 2 implements consensus in the ES environment. The idea of the algorithm is to ensure safety by waiting until a value is contained in every message received in a round. In this way, one can ensure that a value has also been relayed by the current source and is therefore known by everybody (we say that the value is written). If a process evaluates Line 9 to true, then \( \text{VAL} \) is known by everybody (because it was written in the last round) and no other process will consider another value as written, because only a value which has also been relayed by a source can be in \( \text{WRITTEN} \). But the relayed value of a source would also be in \( \text{PROPOSED} \) at every process.

To guarantee the liveness of the consensus algorithm, we use the fact that eventually, all proposal values in the system are received in every even round by everybody and everybody will select the same maximum in Line 12.

Therefore, everybody will propose the same value in the next round and the algorithm will terminate.

3.1. Analysis

For all local variables \( \text{VAR} \), we denote by \( \text{VAR}_{j} \) the local variable of process \( p_i \) (e.g., \( \text{PROPOSED}_{j} \)). For every variable \( \text{VAR} \), \( \text{VAR}^{k} \) is the value of this variable after process \( p_i \) has executed Line 7 when \( \text{compute} \) has been invoked with parameter \( k \) (i.e., in round \( k \)).

**Lemma 1.** If no process has decided yet and for some \( p_i \), \( v \in \text{WRITTEN}^{k}_{i} \), then every process \( p_j \) that enters round \( k \) has \( v \in \text{PROPOSED}^{k}_{j} \).

**Proof:** If a process \( p_i \) has a value \( v \) in \( \text{WRITTEN}^{k}_{i} \), then \( v \) has been contained in every message, which \( p_i \) has received in round \( k \) (Line 6). This includes the message of the source, since by assumption the source has not yet terminated. But by definition, every other process \( p_j \) that enters round \( k \) also has received the message of this source in this round and added it to its set \( \text{PROPOSED}^{k}_{j} \) (Line 7). Therefore, \( v \) is in \( \text{PROPOSED}^{k}_{j} \).

**Lemma 2.** If no process has decided yet and \( p_i \) has \( v \in \text{WRITTENOLD}^{k}_{i} \) in an even round \( k \), then every other process \( p_j \) that enters round \( k \) has \( v \in \text{WRITTEN}^{k}_{j} \).

**Proof:** If a process \( p_i \) has a value \( v \) in \( \text{WRITTENOLD}^{k}_{i} \), then it has had \( v \) in \( \text{WRITTEN}^{k-1}_{i} \). Therefore, every other process \( p_j \) that enters round \( k \) has \( v \) in \( \text{PROPOSED}^{k-1}_{j} \) in the same odd round \( k - 1 \) (Lemma 1). Since no value is removed from a set \( \text{PROPOSED} \) in odd rounds, \( v \) will be contained in every set \( \text{PROPOSED} \) broadcast at the end of round \( k - 1 \) and therefore get into \( \text{WRITTEN}^{k}_{j} \) at every process \( p_j \) that enters round \( k \).

**Theorem 1.** Algorithm 2 implements consensus in the ES environment.

**Proof:** We have to prove the 3 properties of consensus. Validity is immediately clear, because \( \text{VAL} \) is always an initial value.
4. Implementing consensus in ESS

Algorithm 3 implements consensus in the ESS environment. For the safety part, the algorithm is very close to algorithm 2 (see Section 3).

To guarantee liveness, we use the fact that we have at least one process which is eventually a source forever. We use the idea of the construction of the leader failure detector \( \Omega \) [3]. It elects a leader among the processes which is eventually stable. In “known” networks, with some eventual synchrony, \( \Omega \) can be implemented by counting heartbeats of processes (e.g., in [1]). But we are not able to count heartbeats of different processes here, because in our model the processes have no IDs. To circumvent this problem, we identify processes with the history of their proposal values. If several processes have the same history, they either propose the same value, or their histories diverge and will never become identical again. Eventually, all processes will select the same history as maximal history and the processes with this history will propose in every round the same value.

The processes, the eventual source periodically sends a “classical” leader.

However, the history of a process permanently grows. Therefore, every process includes its current history in every message it broadcasts. Furthermore, it maintains a counter \( C \) for every history it has yet heard of (in such a way that no memory is allocated for histories it has not yet heard of). Then, it compares the histories it receives with the ones it has received in previous rounds. If some old history is a prefix of a new history, it assigns the counter of the new history the value of the counter of the old one, increased by one. Thus, the counter of a history that corresponds to an eventual source is eventually increased in every round.

In this way, it is possible to ensure that eventually only eventual sources that converge to the same infinite history consider itself as leader. In a classical approach, eventually only these leaders would propose values. But to meet our safety requirements, it is crucial to ensure that all processes propose in every round at least something to make sure that the value of the current source is received by everybody. Therefore, we let processes that do not consider itself as a leader propose the special value \( \bot \).

4.2. Analysis

Similarly to Section 3, for every variable \( \text{VAR}_i \), \( \text{VAR}_i^k \) is the value of this variable after process \( p_i \) has executed Line 9 in round \( k \).

**Definition 1.** We say, that \( p_i \) has heard of \( p_j \)'s round \( k \) message \( (m_j^k) \), if \( p_i \) has received \( m_j^k \) in round \( k \), or if there exists another process \( p_l \) such that \( p_i \) has heard of \( p_l \)'s round \( k \) message for some \( k' > k \) and \( p_l \) has heard of \( p_j \)'s round \( k \) message.

Let process \( p_s \) be an eventual source. We then identify three groups of processes:

- **out-connected:** The processes, the eventual source \( p_s \) has infinitely often heard of.
- **o-proposer:** The out-connected processes that have eventually in every round timely links towards all other out-connected processes.\(^2\)
- **leader:** We say that a process \( p_i \) is a leader in some round \( k \) (\( p_i \in \text{leader}(k) \)), iff \( \forall H. C_i^k[\text{HISTORY}_i^k] \geq C_i^k[H] \).

If process \( p_i \) is eventually a leader forever, i.e., there exists a \( k \), such that for every round they require only finite space. Thus, if we could ensure that eventually all processes that propose have in every round the same history (and at least one process proposes infinitely often), then the proposal values sent are indistinguishable from the proposal values of a single “classical” leader.

\(^2\) Note that it is possible that the message an out-connected process actually has received is not the message that a \( o \)-proposer has sent. It is sufficient if it receives an identical message from another process.
Lemma 3. \( \diamond \) corresponds to a maximal counter at a \( \diamond \).

Algorithm 3: The consensus algorithm in ESS for process \( p_i \).

1. on initialization do
2. \( \text{VAL} := \text{initial value}; \forall H, C[H] := 0; \text{HISTORY} := \text{VAL}; \)
3. \( \text{WRITTEN} := \text{WRITTENOLD} := \text{PROPOSED} := 0; \)
4. return \( m = (\text{PROPOSED}, \text{HISTORY}, C); \)
5. on compute\((k, M_i)\) do
6. \( \text{WRITTEN} := \bigcap_{m \in M_i[k_i]} m.\text{PROPOSED}; \)
7. \( \text{PROPOSED} := (\bigcup_{m \in M_i[k_i]} m.\text{PROPOSED}) \cup \text{PROPOSED}; \)
8. \( \forall H, C[H] := \min_{m \in M_i[k_i]} (m.C[H]); \)
9. \( \forall m \in M_i[k_i], C[m.\text{HISTORY}] := 1 + \max \{ C[H] \mid H \text{ is a prefix of } m.\text{HISTORY} \}; \)
10. if \( (k, i \mod 2 = 0) \) then
11. \( \text{if } (\text{WRITTEN} = \{\text{VAL}\}) \land (\text{PROPOSED} \subseteq \{\text{VAL}, \bot\}) \text{ then} \)
12. \( \quad \text{decide } \text{VAL}; \text{halt}; \)
13. else if \( (\text{WRITTEN} \setminus \{\bot\} \neq 0) \) then
14. \( \quad \text{VAL} := \max(\text{WRITTEN} \setminus \{\bot\}); \)
15. \( \text{if } (\forall H, C[\text{HISTORY}] \geq C[H]) \lor (\text{PROPOSED} \not\subseteq \{\text{VAL}, \bot\}) \text{ then} \)
16. \( \quad \text{PROPOSED} := \{\text{VAL}\}; \)
17. else
18. \( \quad \text{PROPOSED} := \{\bot\}; \)
19. \( \text{WRITTENOLD} := \text{WRITTEN}; \)
20. \( \text{WRITTEN} := \text{PROPOSED}; \)
21. append \text{VAL} to \text{HISTORY};
22. return \( m = (\text{PROPOSED}, \text{HISTORY}, C); \)

Lemma 7. If no process has decided yet, then eventually only values of leaders and \( \bot \) get into a set \text{WRITTEN} anywhere.

Theorem 2. Algorithm 3 implements consensus in ESS.

Proof: We have to prove the 3 properties of consensus. Validity is clear, since \text{VAL} is always an initial value.

To prove termination, assume there exists a run where no process ever decides. Then, eventually only non-\( \bot \) values of leaders will get into a set \text{WRITTEN} anywhere (Lemma 7) and they will get into \text{WRITTEN} always in every even round (Lemma 3) and all out-connected processes select the same value (the maximum in Line 14). Therefore, only this value and \( \bot \) will be written in subsequent rounds and every out-connected process will select this value as value for \text{PROPOSED} in Line 16 (i.e., no out-connected process will select \( \bot \)) and everybody will evaluate Line 11 to true in the next round. Therefore, eventually, every correct process will decide.

To prove agreement, assume \( p_i \) is the first process that decides a value \( v \) in a round \( k \). This means, that \( p_i \) has evaluated Line 11 to true. Then, as \( \text{PROPOSED} \subseteq \{v, \bot\} \), no other value different from \( \bot \) is in a set \text{WRITTEN} anywhere in the system (compare Lemma 1) and \( v \) is in \text{WRITTEN} at every process in the system in round \( k \), since it is also in \text{WRITTENOLD} (compare Lemma 2). Therefore, every other process decides \( v \) in the same round, or \( v \) in the same round and \( v \) as new \text{VAL} and no other value different from \( \bot \) will ever get into \text{PROPOSED} anywhere in the system and therefore, no other value will ever be selected as \text{VAL}.

\[ \Box \]

5. Weak-Sets

The weak-set data structure has been introduced by Delporte-Gallet and Fauconnier in [4].

A weak-set \( S \) is a shared data structure that contains a set of values. It is defined by two operations: the \text{addS}(v) operation to add a value \( v \) to the set and the \text{getS} operation which returns a subset of the values contained in the weak-set. Note that we do not consider operations to remove values from the set. Every \text{getS} operation returns all values \( v \) where the corresponding \text{addS}(v) operation has completed before the beginning of the \text{getS} operation. Furthermore, no value \( v' \) where no \text{addS}(v') has started before the termination of the \text{getS} operation is returned. For \text{addS} operations concurrent with the \text{getS} operation, it may or may not return the values. Therefore, weak-sets are not necessarily linearizable\(^3\).

5.1. Weak-Sets and registers

A weak-set is clearly stronger than a (regular) register:

3. A weak object is linearizable (also called atomic) if all of its operations appear to take effect instantaneously [11].
Lemma 8. If for some $p_i$ with parameter $p$, executed Line 15 in round $k$, every set PROPOSED by all correct processes, every value will eventually be in Line 11. But since eventually all messages will be received at all correct processes and that every get operation returns all writes have completed, the return value will be the same at all processes. To see that also validity holds, consider the two different processes may return different values, but after follows directly from the termination property of weak-sets.

If several processes write at the same time, two reads at two different reads processes may return different values, but after all writes have completed, the return value will be the same at all processes. To see that also validity holds, consider the value returned by a read. If there is no concurrent write, then the value returned is the last value written (i.e. the maximal value of all values concurrently written).

\[ \Box \]

In [4], a weak-set is implemented using (atomic) registers in the following two cases:

**Proposition 2.** If the set of processes using the weak set is known (i.e. the IDs and the quantity), then weak-sets can be implemented with single-writer multiple-reader registers.

**Proposition 3.** If the set of possible values for the weak set is finite, then weak-sets can be implemented with multi-writer multiple-reader registers.

### 5.2. Weak-Sets and the MS environment

Algorithm 4 shows how to implement a weak-set in the MS environment. Similarly to Section 3, for every variable $VAR_i$, $VAR_i^k$ is the value of this variable after process $p_i$ has executed Line 15 in round $k$ (i.e. after compute is called with parameter $k$).

**Lemma 8.** If for some $p_i$, $v \in \text{WRITTEN}^k$, then every process $p_j$ that enters round $k$ has $v \in \text{PROPOSED}^k$.

**Proof:** The proof is analogous to Lemma 1.

**Lemma 9.** If some value is in $\text{WRITTEN}$ at some process, then this value will be forever in $\text{PROPOSED}$ at all processes.

**Proof:** Since it is never a value removed from any set $\text{PROPOSED}$, this follows immediately from Lemma 8.

**Theorem 3.** Algorithm 4 implements a weak-set.

**Proof:** We have to show that all operations terminate at all correct processes and that every get operation returns all values which have been added before.

The only position where an operation may be blocked is in Line 11. But since eventually all messages will be received by all correct processes, every value will eventually be in every set $\text{PROPOSED}$ and therefore eventually be in every set $\text{WRITTEN}$. Thus, no correct process will block in Line 11 forever.

To show that every get operation returns all values which have been added before, see that an add($v$) operation only terminates if $v$ is in $\text{WRITTEN}$ at some process. Together with Lemma 9, this means that this value will be returned by every process in Line 6.

\[ \Box \]

### 5.3. Emulation of the MS environment with weak-sets

Algorithm 5 emulates the MS environment using a weak-set $S$ and the corresponding $\text{add}_S$ and $\text{get}_S$ methods.

As a weak-set is implementable by only using registers (see Proposition 2) and the FLP impossibility result [8] states that consensus is not implementable using only registers, this implies, that it is not possible to implement consensus in the MS environment (without any additional assumptions like in ES).

**Algorithm 4:** A weak-set algorithm in the MS environment for process $p_i$.

1. on initialization do
2. \hspace{1em} VAL := ⊥; PROPOSED := WRITTEN := ∅;
3. \hspace{1em} BLOCK := false;
4. \hspace{1em} return PROPOSED;
5. on get do
6. \hspace{1em} return PROPOSED;
7. on add($v$) do
8. \hspace{1em} PROPOSED := PROPOSED ∪ \{v\};
9. \hspace{1em} VAL := v;
10. \hspace{1em} BLOCK := true;
11. \hspace{1em} wait until (BLOCK = false);
12. \hspace{1em} return ack;
13. on compute($k_i, M_i$) do
14. \hspace{1em} WRITTEN := \bigcap_{m \in M_i[k_i]} m;
15. \hspace{1em} PROPOSED := (\bigcup_{m \in M_i[k_i], 1 \leq k' \leq k, m} m) \cup PROPOSED;
16. \hspace{1em} if (VAL ∈ WRITTEN) then BLOCK := false;
17. \hspace{1em} return PROPOSED;

**Algorithm 5:** Emulating the MS environment for process $p_i$ using a weak-set $S$.

1. on initialization do
2. \hspace{1em} DELIVERED := ∅;
3. \hspace{1em} trigger end-of-round;
4. on send($m_i, k_i$) do
5. \hspace{1em} \text{adds}((m_i, k_i));
6. \hspace{1em} forall $(m, k) \in \text{get}_S \setminus \text{DELIVERED}$ do
7. \hspace{2em} DELIVERED := DELIVERED ∪ \{(m, k)\};
8. \hspace{1em} trigger receive($m, k$);
9. \hspace{1em} trigger end-of-round;
Theorem 4. Algorithm 5 emulates the MS environment.

Proof: Clearly, eventually all messages get delivered and all correct processes execute an infinite number of rounds.

It remains to show, that in every round \( k \), there exists a process \( s_k \) such that for every process \( p_i \) at which end-of-round, occurs in round \( k \), \( p_i \) receives the round \( k \) message of \( s_k \) in round \( k \).

Let \( p_1 \) be the first process that finishes to add the value of a round \( k \). If several processes finish to add their values at exactly the same time, choose one.

Claim: Every process at which end-of-round is triggered in round \( k \) has received \( p_1 \)'s round \( k \) value.

The proof is by contradiction. Assume that a process \( p_j \) triggers end-of-round in round \( k \) without having received \( p_1 \)'s round \( k \) value. By the definition of a weak-set, this means that \( p_j \)'s get \( _S \) begun before \( p_1 \)'s add \( _S \) was completed. But a process will only start a get \( _S \) after it has finished to add its own value. A contradiction to the fact that \( p_1 \) was the first process that has completed its add \( _S \).

6. The MS-environment and the \( \Sigma \) failure detector

The quorum failure detector \( \Sigma \) [5] outputs lists of IDs of trusted processes (i.e. it is not well-defined in our anonymous model) and it satisfies the following properties:

Intersection: Given any two lists of trusted processes, possibly at different times and by different processes, at least one process belongs to both lists.

Completeness: Eventually at all correct processes, every trusted process is correct.

\( \Sigma \) has been shown to be the weakest failure detector to emulate registers in totally asynchronous message-passing systems [5] (with known IDs). This means, that \( \Sigma \) is sufficient to emulate registers in such systems and with any failure detector which is also sufficient to implement registers in such a system, it is possible to emulate \( \Sigma \). Interestingly, although it is possible to implement a register in the MS environment (via weak-sets), we show that even if we assume that the number of processes and their IDs are known, it is not possible to emulate \( \Sigma \). Note that this is no contradiction, since in our model no failure detector is necessary for the emulation.

Proposition 4. It is not possible to emulate \( \Sigma \) in the MS-environment, even if the number of processes and their IDs are known.

Proof: Assume there exists such an algorithm and consider a run \( r_1 \) where process \( p_1 \) is the only correct process, \( p_1 \) is always the source, and \( p_1 \) receives no messages from other processes. Then, by the completeness property of \( \Sigma \), there exists some time \( t \) after which the output of \( \Sigma \) is \( \{ p_1 \} \).

Similarly, consider a run \( r_2 \) where process \( p_2 \) is the only correct process and \( p_1 \) crashes after time \( t \). Again, \( p_1 \) is the source until time \( t \) and receives no messages from other processes (this is possible, since the messages from \( p_2 \) may be arbitrary delayed). For \( p_1 \), run \( r_1 \) and \( r_2 \) are indistinguishable up to time \( t \) and consequently the \( \Sigma \) at \( p_1 \) will output \( \{ p_1 \} \) at \( p_1 \) at time \( t \). But since eventually, the output at \( p_2 \) has to be \( \{ p_2 \} \) forever, this contradicts to the intersection property of \( \Sigma \).

7. Conclusions

This paper has provided algorithms to emulate registers and solve consensus under different synchrony assumptions in unknown and anonymous message-passing networks that suffer from crash-failures. One of these algorithms uses a novel pseudo leader election primitive.

Furthermore, we have shown that the MS environment (i.e. a system with a moving timely source) is equivalent to weak-sets, a generalization of registers for unknown and anonymous systems. In some sense, this indicates that the synchrony assumptions in this environment are necessary to implement basic safety primitives.

Additionally, we have shown that in the MS environment, it is not possible to emulate \( \Sigma \), the weakest failure detector to emulate registers [5], even if we assume the existence of IDs and a bound on the number of processes. To the best of our knowledge, we found for the first time a partially synchronous environment in which registers are implementable and \( \Sigma \) is not.

References


