Packet Efficient Implementation of the Omega Failure Detector

Quentin Bramas1 · Dianne Foreback2 · Mikhail Nesterenko2 · Sébastien Tixeuil3

Published online: 23 February 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract We assume that a message may be delivered by packets through multiple hops and investigate the feasibility and efficiency of an Omega Failure Detector implementation. To motivate the study, we prove the existence and sustainability of a leader is exponentially more probable in a multi-hop than in a single-hop implementation. An implementation is: message efficient if all but finitely many messages are sent by a single process; packet efficient if it is message efficient and the number of packets used to transmit all but finitely many messages is proportional to the number of processes in the system; super packet efficient if it is message efficient and the number of channels used to transmit all but finitely many packets is proportional to the number of processes in the system. We prove that a super packet efficient implementation of Omega is impossible. We establish necessary conditions for the
existence of a packet efficient implementation of Omega and present an algorithm that implements Omega under these conditions.

Keywords Failure detectors · Fault tolerance · Consensus · Distributed algorithms

1 Introduction

The asynchronous system model places no assumptions on message propagation delay or relative process speeds. This makes the model attractive for distributed algorithm research as the results obtained in the model are applicable to an arbitrary network and computer architecture. However, the fully asynchronous system model is not well suited for fault tolerance studies. An elementary problem of consensus, where processes have to agree on a single value, is unsolvable even if only one process may crash [16]: the asynchrony of the model precludes processes from differentiating a crashed and a slow process.

A failure detector [8] is a construct that enables the solution to consensus or related problems in this model. Potentially, a failure detector may be very powerful and, therefore, hide the solution to the problem within its specification. Conversely, the weakest failure detector specifies the least amount of synchrony required to implement consensus [7]. One such detector is Omega.1

Naturally, a failure detector may not be implemented in the asynchronous model itself. Hence, a lot of research is focused on providing the implementation of a detector, especially Omega, in the least restrictive communication model. These restrictions often deal with timeliness and reliability of message delivery. Aguilera et al. [2] provide a remarkable Omega implementation that requires only a single process to have eventually timely channels to the other processes and a single process to have so called fair-lossy channels to and from all other processes. Aguilera et al. present what they call an efficient implementation where only a single process sends infinitely many messages. In their work, they consider a direct channel as the sole means of message delivery from one process to another. In this paper, we study a more general setting where a message may arrive at a particular process through several intermediate processes.

1.1 Our Contribution

We study an Omega implementation under the assumption that a message may be transmitted to its destination through other processes.

To motivate this multi-hop Omega implementation approach, we consider a fixed probability of channel timeliness and study the probability of leader existence in a classic single-hop and in multi-hop implementations. We prove that, as the network

---

1In literature, the detector is usually denoted by the Greek letter. However, we use the letter to denote the complexity lower bound. To avoid confusion, we spell out the name of the failure detector in English.
size grows, the probability of leader existence tends to zero for a single-hop implementation and to one for a multi-hop one. Moreover, if the timeliness of a channel changes, the probability of a leader persisting tends to zero for a single-hop and to one for a multi-hop implementation.

We then consider deterministic algorithms and study three classes of Omega implementations: message efficient, packet efficient and super packet efficient. In a message efficient implementation all but finitely many messages are sent by a single process. A packet efficient implementation is message efficient, and for all but finitely many messages, the number of packets is linear with respect to the number of processes in the network. A super packet efficient implementation adds an additional restriction that the packets of all but finitely many messages have to use a linear number of channels. In other words, in the (simple) packet efficient implementation, packets of different messages may use different channels; potentially all channels of the system are periodically used. In a super packet efficient implementation, the packets have to use the same and linear number of channels.

We establish conditions necessary for these implementations. The conditions relate to the timeliness and reliability of packet delivery. Let us give informal introductions to these conditions, a more formal treatment is carried out in the subsequent sections. A packet is timely if it is delivered within a bounded number of steps. A packet is reliable if it is delivered at all. A channel is eventually timely if it delivers only finitely many non-timely packets. In a fair-lossy channel, if the same packet is sent infinitely many times, it is delivered infinitely often. To put this another way, a fair-lossy channel may lose the same packet only finitely many times.

In this paper, we determine that if the timeliness of the packets of one message does not correlate with the timeliness of another, i.e., there are no timely channels, then a packet efficient implementation of Omega is impossible. If eventually timely and fair-lossy channels are allowed, we establish the necessary and sufficient conditions for the existence of a packet efficient implementation of Omega. We present an algorithm that uses these necessary conditions and provides a message and packet efficient implementation of Omega. We prove that a super packet efficient implementation is impossible altogether.

Our algorithm can operate without modifications in a network that is not completely connected so long as the timeliness and reliability requirements are met. We conclude the paper with this and other extensions to our algorithm.

1.2 Related Work

The failure detector implementation is a well-researched area [1, 3, 5, 12, 14, 18, 20–24, 26]. We are limiting our literature review to the studies that are most recent and closest to our work.

Delporte-Gallet et al. [12] describe algorithms for recognizing timely channel graphs. Their algorithms are super packet efficient and may potentially be used to implement Omega. However, their solutions assume non-constant size messages and perpetually reliable channels. That is, Delporte-Gallet et al. deviate from the model of Aguilera et al. and the algorithms of Delporte-Gallet et al. do not operate correctly under fair-lossy and eventually timely channel assumptions.
A number of papers also consider an Omega implementation under various modifications of the Aguilera et al. model. Hutle et al. [18] implement Omega assuming a send-to-all message transmission primitive where \( f \) processes are guaranteed to receive the message timely. Fernandez and Raynal [3] assume that a process is able to timely deliver its message to a quorum of processes over direct channels. This quorum and channels may change with each message. A similar rotating set of timely channels is used by Malkhi et al. [22]. Larrea et al. [21] give a message efficient implementation of Omega but assume that all channels are eventually timely. In their Omega implementation, Mostefaoui et al. [23] rely on a particular order of query and response message interleaving rather than on the timeliness of messages. A hybrid implementation by Mostefaoui et al. [24] combines the timely approach of [1] and the time-free approach of [23]. It requires messages from a single process \( p \) be received by a set of \( f \) processes, in any combination of timely as in [23], or be among the first \((n - f)\) replies to each of the \( f \) processes’ alive query. Biely and Widder [5] also consider a message-driven (i.e., non-timer based) model and provide a message efficient Omega implementation. Delporte-Gallet et al. [11] give message efficient pseudo-stabilizing and self-stabilizing algorithms. In their implementations, pseudo-stabilizing algorithms are given for a model that assumes every channel is fair and there is at least one process with timely outgoing channels, and another in a model that assumes there is at least one process with timely incoming and outgoing channels. They also provide a self-stabilizing algorithm when the model is strengthened with all-timely channels.

There are several recent papers on timely solutions to problems related to Omega implementations. Charron-Bost et al. [10] use a timely spanning tree to solve approximate consensus. Lafuente et al. [20] implement the eventually perfect failure detector using a timely cycle of processes. Fernández-Campusano et al. [15] give an efficient algorithm to implement a modified definition of Omega in a crash-recovery model with eventually timely channels among a set of core processes that eventually do not crash.

There are several papers presenting Omega implementations in a model where the number of processes are not known but the processes are totally ordered by unique process identifiers. Although not efficient, Jimenez et al. [19] implement Omega in a model with unreliable channels and an eventually timely path from the leader process to all other processes. Fernández et al. [14] provide an efficient implementation of Omega that requires a pair of uni-directional, fair-lossy channels connected to each pair of processes with a single process having fair-lossy output channels to all other correct processes. Arantes et al. [4] propose an implementation of Omega in a network whose nodes are mobile and the topology changes.

2 Notation and Definitions

2.1 Differences with the Aguilera Model

To simplify the presentation, we use an even more general model than what is used in Aguilera et al. [2]. For the reader who is familiar with their model, we quickly outline these differences before presenting the complete model. We use infinite capacity
non-FIFO channels rather than single packet capacity channels. Our channel construct makes us explicitly state the packet fairness propagation assumptions that are somewhat obscured by the single capacity channels.

We do not differentiate between a slow process and a slow channel since slow channels may simulate both. Our Omega implementation code is expressed in terms of guarded commands, rather than the usual more procedural description. The operation of our algorithm is a computation, which is a sequence of these command executions. We express timeouts directly in terms of computation steps rather than abstract or concrete time. This simplifies reasoning about them.

Despite the differences, the models are close enough such that all of the results in this paper are immediately applicable to the traditional Omega implementation model.

2.2 Processes and Computations

A computer network consists of a set \( N \) of processes. The cardinality of this set is \( n \). Each process has a unique identifier from 0 to \( n - 1 \). Processes interact by passing messages through non-FIFO unbounded communication channels. Each process has a channel to all other processes. That is, the network is fully connected. A message is constant size if the data it carries is in \( O(\log n) \). For example, a constant size message may carry several process identifiers but not a complete network spanning tree. See Peleg [25] for more details on this assumption.

Each process has variables and actions. The action has a guard: a predicate over the local variables and incoming channels of the process. An action is enabled if its guard evaluates to true. A computation is a potentially infinite sequence of global network states such that each subsequent state is obtained by executing an action enabled in the previous state. This execution is a computation step. Processes may crash. A crashed process stops executing its actions. A correct process does not crash. See Chandy and Misra [9] for more details on the guarded command language.

2.3 Messages and Packets

We distinguish between a packet and a message. Message is particular content to be distributed to processes in the network. Origin is the process that initiates the message. The identifier of the origin is included in the message. Messages are sent via packets. Packet is a portion of data transmitted over a particular channel. A message is the payload of a packet. A process may receive a packet and either forward the message it contains or not. A process may not modify it: if a process needs to send additional information, the process may send a separate message. A process may forward the same message at most once. In effect, a message is transmitted to processes of the network using packets. A particular process may receive a message either directly from the origin, or indirectly possibly through multiple hops.

2.4 Scheduling and Fairness

We express process synchronization in terms of an adversarial scheduler. The scheduler forms a computation of an algorithm by arbitrarily interleaving enabled action
executions that are subject to the following constraints. We do not distinguish slow processes and slow packet propagation. A scheduler may express these phenomena through scheduling process action execution in a particular way. A packet transmission immediately enables the packet receipt action in the recipient process. A packet is lost if the receipt action is never executed. A packet is not lost if it is eventually received.

### 2.5 Timers

Timer is a construct with the following features. It can be in either of the two states: on or off. It has a **timeout integer** value and a **timeout action** associated with it.

A timer is either a receiver timer or a sender timer. If a **sender timer** is on, then within timeout integer steps, either the timer is reset or the timeout action is executed. If a **receiver timer** is on, then after timeout integer steps, either the timer is reset or the timeout action is executed. Note that the sender timer guarantees timeout action execution within a certain number of steps while the receiver timeout fires only after a certain number of steps. Thus, if the sender and receiver are connected by a timely path and the sender and receiver timeout integers are properly set, the sender may send messages triggered by the sender’s timeout action and the receiver may receive them without triggering the receiver’s timeout action.

A timer may be reset or stopped. A timer may be checked to determine if it is on or off. Resetting the timeout turns it on. Its timeout integer may be increased. Increasing this timeout integer adds an arbitrary positive integer value to it.

### 2.6 Reliable and Timely Messages and Packets

A packet is **reliable** if it is received, it is **unreliable** otherwise. A message is reliable if it is received by every correct process; i.e. one that does not crash. A channel is (perpetually) reliable if every packet transmitted over this channel is reliable. A channel is eventually reliable if the number of unreliable packets it transmits is finite. By this definition a channel that transmits a finite number of packets is always eventually reliable.

A channel is **fair-lossy** if it has the following properties. If there is an infinite number of packet transmissions over a particular fair-lossy channel of a particular message type and origin, then infinitely many are received. We assume that a fair-lossy channel is not type discriminating. That is, if it is fair-lossy for one type and origin, it is also fair-lossy for every pair of message type and origin.

Observe that if there is an infinite number of message transmissions of a particular message type and origin over a path that is fair-lossy, then infinitely many succeed. The converse is true as well: if there is an infinite number of successful message transmissions, there must be a fair-lossy path between the origin and the destination.

A packet is **timely** if it is received within a bounded number of computation steps. Specifically, there exists a finite integer $B$ such that the packet is received within $B$ steps. This bound may be different for different packets and may not be known to the algorithm. Naturally, a timely packet is a reliable packet. A message is timely if it
is received by every process via a path of timely packets. A channel is (perpetually) timely if every packet transmission over this channel is timely. A channel is eventually timely if the number of non-timely packets it transmits is finite. Thus, a channel that transmits a finite number of packets is always eventually timely.

The timely channel definition is relatively clear. The opposite, the non-timely channel, is a bit more involved. A channel that occasionally delays or misses a few packets is not non-timely as the algorithm may just ignore the missed packets with a large enough timeout. Hence, the following definition.

A channel is strongly non-timely if the following holds. If there is an infinite number of packet transmissions of a particular type and origin over a particular non-timely channel, then, for any fixed integer, there are infinitely many computation segments of this length such that none of the packets are delivered inside any of the segments.

Similarly, the non-timeliness has to be preserved across multiple channels, a message may not gain timeliness by finding a parallel timely path, then, for example, the two paths may alternate delivering timely messages. Therefore, we add an additional condition for non-timeliness.

All paths between a pair of processes $x$ and $y$ are strongly non-timely if $x$ sends an infinite number of messages to $y$, yet regardless of how the message is forwarded or what path it takes, for any fixed integer, there are infinitely many computation segments of this length such that none of the messages are delivered inside any of the segments. Hereafter, when we discuss non-timely channels and paths, we mean strongly non-timely channels and paths.

### 2.7 Communication Models

To make it easier to address the variety of possible communication restrictions, we group them into two communication models that we study in this paper. The general propagation model does not allow either reliable or timely channels. Thus, the success and timeliness of two packet transmissions across a particular channel are not related.

In contrast, the dependable (channel) model allows eventually and perpetually reliable timely or fair-lossy channels. In the dependable model, an algorithm may potentially discover the dependable channels by observing packet propagation.

### 2.8 Message Propagation Graph

Message propagation graph is a directed graph over network processes and channels that determines whether packet propagation over a particular channel would be successful. This graph is connected and has a single source: the origin process. This concept is a way of reasoning about scheduling of packets for a particular message.

Each message has two propagation graphs. In reliable propagation graph $R$, each edge indicates whether the packet is received if transmitted over this channel. In timely propagation graph $T$, each edge indicates whether the packet is timely if transmitted over this channel. Since a timely packet is a reliable packet, for the same message, the timely propagation graph is a subgraph of the reliable propagation
graph. In general, a propagation graph for each message is unique. That is, even for the same source process, the graphs for two messages may differ. This indicates that different messages may take divergent routes.

If a channel from process $x$ to process $y$ is reliable, then edge $(x, y)$ is present in the reliable propagation graph for every message where process $x$ is present. In other words, if the message reaches $x$ and $x$ sends it to $y$, then $y$ receives it. A similar discussion applies to a timely channel and the corresponding edges in timely propagation graphs.

Propagation graphs are determined by the scheduler in advance of the message transmission. That is, the recipient process, depending on the algorithm, may or may not forward the received message along a particular outgoing channel. However, if the process forwards the message, the presence of an edge in the propagation graph determines the success of the message transmission. Note that the process forwards a particular message at most once. Hence, the propagation graph captures the complete possible message propagation pattern. A process may crash during message transmission. This crash does not alter propagation graphs.

**Definition 1** A message is reliable only if its reliable propagation graph $R$ is such that every correct process is reachable from the origin through non-crashed processes.

**Definition 2** A message is timely only if its timely propagation graph $T$ is such that every correct process is reachable from the origin through non-crashed processes.

### 2.9 Omega Implementation and its Efficiency

An algorithm that implements the Omega Failure Detector (or just Omega) is such that in a suffix of every computation, each correct process outputs the identifier of the same correct process. This process is the leader.

An implementation of Omega is message efficient if the origin of all but finitely many messages is a single correct process and all but finitely many messages are constant size. An implementation of Omega is packet efficient if all but finitely many messages are transmitted using $O(n)$ packets.

An Omega implementation is super packet efficient if it is packet efficient and the packets of all but finitely many messages are using the same channels. In other words, if a packet of message $m_1$ is forwarded over some channel, then a packet of another message $m_2$ is also forwarded over this channel. The intent of a super packet efficient algorithm is to only use a limited number of channels infinitely often. Since a packet efficient algorithm uses $O(n)$ packets infinitely often, a super packet efficient algorithm uses $O(n)$ channels infinitely often.

### 3 Probabilistic Properties

In this section, we contrast a multi-hop implementation of Omega and a classic single-hop, also called direct channel, implementation in a model where channel timeliness is probabilistic. We assume each network channel is timely with probability $p$. The
timeliness probabilities of any two channels are independent. A chain of timely channels forms a timely path.

### 3.1 Leader Existence Probability

We assume that the leader may exist only if there is a process that has timely paths to all processes in the network. In case of a direct channel implementation, the length of each such path must be exactly one. We do not consider models where the leader may be established without timely communication to all nodes.

As \( n \) grows, Omega implementations behave radically differently. Theorems 1 and 2 state the necessary conditions for leader existence and indicate that the probability of leader existence for a direct channel implementation approaches zero exponentially quickly, while this probability for a multi-hop implementation approaches one exponentially quickly. In practical terms, a multi-hop omega implementation is far more likely to succeed in establishing the leader.

**Theorem 1** If the probability of each channel to be timely is \( p < 1 \), then the probability of leader existence in any direct channel Omega implementation approaches zero exponentially fast as \( n \) grows.

**Proof** Let \( D_x \) be an event to which probability \( \mathbb{P}(D_x) \) is assigned. This represents the probability that some process \( x \) does not have direct timely channels to all processes in the network. This probability is \( \mathbb{P}(D_x) = 1 - p^{n-1} \). For two distinct processes \( x \) and \( y \), \( D_x \) and \( D_y \) are disjoint since channels are oriented. Thus, if \( p < 1 \), the probability that no leader exists is \( \mathbb{P}(\bigcap_{x \in V} D_x) = (1 - p^{n-1})^n \sim 1 - p^n. \)

**Theorem 2** If the probability of each channel to be timely is \( p < 1 \), then the probability of leader existence in any multi-hop Omega implementation approaches 1 exponentially fast as \( n \) grows.

**Proof** A channel is *bitimely* if it is timely in both directions. The probability that there exists at least one process such that there exist timely paths from this process to all other processes is greater than the probability to reach them through bitimely paths. We use the probability of the latter as a lower bound for our result. If \( p \) is the probability of a channel to be timely, \( \tilde{p} = p^2 \) is the probability that it is bitimely. Consider graph \( G \) where the edges represent bitimely channels. It is an Erdos-Renyi graph where an edge exists with probability \( \tilde{p} \). It was shown (see [17]) that \( \mathbb{P}(G \text{ is connected}) \sim 1 - n(1 - \tilde{p})^{n-1} \sim 1 - p^n. \)

### 3.2 Leader Stability

As in the previous subsection, we assume the leader has timely paths to all other processes in the network. However, if the timeliness of each channel changes, this process may not have timely paths to all other processes anymore. Leader stability

---

\(^2x \sim y \) means that \( x \) approaches \( y \) as \( n \to \infty \). See [17] for usage.
time is the expected number of rounds of such channel timeliness change where a particular process remains the leader.

Again, direct channel and multi-hop implementations of Omega behave differently. Direct channel leader stability time approaches zero as \( n \) increases and cannot be limited from below by fixing a particular value of channel timeliness probability. Multi-hop leader stability goes to infinity exponentially quickly. In a practical setting, a leader is significantly more stable in a multi-hop Omega implementation than in a direct channel one.

**Theorem 3** In any direct channel Omega implementation, if the probability of each channel to be timely is \( p < 1 \), leader stability time goes exponentially fast to 0 as \( n \) grows. If leader stability time is to remain above a fixed constant \( E > 0 \), then the channel timeliness probability \( p \) must converge to 1 exponentially fast as \( n \) grows.

**Proof** At a given time, a given process has timely channels to all other processes with probability \( p^{n-1} \). The number of rounds \( X \) a given process retains timely paths to all other processes follows a geometric distribution \( \mathbb{P}(X = r) = q^r(1-q) \), where \( q = p^{n-1} \). Thus, the expected number of rounds a process retains timely channels to all other processes is \( \frac{q}{1-q} = \frac{p^{n-1}}{1-p^{n-1}} \sim p^{n-1} \), which tends exponentially fast toward 0 if \( p \) is a constant less than 1.

Assume \( \mathbb{E}(X) \) converges toward a given fixed number \( E \) as \( n \) tends toward infinity. That is, we need \( \lim_{n \to \infty} \mathbb{P}(G \text{ is connected}) = \frac{1}{E+1} \). Then, \( p^{n-1} \) tends to \( \frac{1}{E+2} \), which implies that \( p \) converges toward 1 exponentially fast.

**Theorem 4** In any multi-hop Omega implementation, if the probability of each channel to be timely is \( p < 1 \), leader stability time goes to infinity exponentially fast as \( n \) grows. If leader stability time is to remain above a fixed constant \( E > 0 \), then channel timeliness probability may converge to 0 exponentially fast as \( n \) grows.

**Proof** If we fix \( \tilde{p}, 0 < \tilde{p} < 1 \), we have \( \mathbb{P}(G \text{ is connected}) \sim 1 - n(1-\tilde{p})^{n-1} \) (see [13, 17]). Then, the expected number of rounds a given process retains timely paths to all other processes is asymptotically \( n^{-1} \left( \frac{1}{1-\tilde{p}} \right)^n \), which increases exponentially fast.

Assume \( \mathbb{E}(X) \) converges toward a given fixed number \( E \) as \( n \) tends to infinity. This means that

\[
\lim_{n \to \infty} \mathbb{P}(G \text{ is connected}) = \frac{1}{E+1} = e^{-e^{-c}}
\]

Using well-known results of random graph theory [6], we can take

\[
\tilde{p}(n) = \frac{\ln n}{n} + \frac{c}{n} = \frac{\ln n}{n} - \frac{\ln (1 + E)}{n}
\]

\[
\lim_{n \to \infty} \tilde{p}(n) = e^{-e^{-c}}
\]

\[
\frac{\ln n}{n} = \frac{c}{n} - \frac{\ln (1 + E)}{n}
\]

\[
\lim_{n \to \infty} \frac{\ln n}{n} = 0
\]

\[
\lim_{n \to \infty} \frac{c}{n} = c
\]

\[
\lim_{n \to \infty} \frac{\ln (1 + E)}{n} = 0
\]

4 Necessity and Sufficiency Properties

We now explore the properties of a deterministic Omega implementation.
4.1 Model Independent Properties

The below Omega implementation properties are applicable to both the general propagation and the dependable channel models. Intuitively, Theorem 5 states that the leader needs to periodically inform other processes of its correctness or they are not able to detect its crash.

Theorem 5 In any implementation of Omega, at least one correct process sends infinitely many timely messages.

Proof Let A be an implementation of Omega and let σ be its computation. Assume all correct processes send finitely many timely messages in σ. Consider a non-trivial case where this computation contains more than one correct process.

Since A is an implementation of Omega, a process is eventually elected the leader in σ by all correct processes. Let it be some process x. By assumption, x sends only finitely many timely messages. Consider a computation σ’ of A, which has the same prefix ρ as σ up to the election of the leader x or the last timely message receipt, whichever comes last. Let x crash in σ’ after this prefix ρ. Since A is an implementation of Omega, after the crash of x, another process y must be elected the leader in σ’ by the remaining correct processes. Consider a computation σ’’ of A that shares the prefix ρ with σ’ but where x does not crash, however, all messages from x are delayed. Since, by assumption, all timely messages are sent in ρ, the remaining messages can be delayed arbitrarily long. This means that in σ’’, y is also elected the leader by all correct processes but x. That is, if A has a computation σ where all correct processes elect x a leader, it also has a computation σ’’ where all correct processes switch their election to a different process.

Continuing this discussion, we observe that A must have a computation where correct processes keep switching their leader indefinitely. However, A is an implementation of Omega. This means that in all of its computations, all correct processes have to, eventually, elect a single leader. This contradiction shows that our initial assumption that A has a computation with finitely many timely messages is incorrect. The theorem follows.

Observe that Theorem 5 implies that each computation of every Omega implementation must be infinite. If a single process sends an infinite number of messages in a message efficient implementation of Omega, this process must be the leader. Indeed, if the leader sends only finitely many messages and then crashes, the other processes may not be able to detect this crash. Hence, the below corollary of Theorem 5.

Corollary 1 In a message efficient implementation of Omega, the leader must send infinitely many timely messages.

4.2 General Propagation Model

The intuition for the results in this subsection is as follows. Since there are no channel dependability properties in the general propagation model, for timely delivery
a message needs to be sent across every channel. Indeed, in case some channel is skipped, this may be the channel that contains the only timely path leading to some process. Skipping this channel precludes timely message delivery. It follows that $\Omega(n^2)$ packets are required to timely deliver a message in the general propagation model. Therefore, there does not exist a message and packet efficient implementation of Omega in this model.

**Lemma 1** To timely deliver a message in the general propagation model, each recipient process needs to send it across every outgoing channel, except for possibly the channels leading to the origin and the sender.

**Proof** Assume the opposite. There exists an algorithm $\mathcal{A}$ that timely delivers message $m$ from the origin $x$ to all processes in the network such that some process $y$ receives it timely yet does not forward it to some process $z \neq x$.

Consider the propagation graph $T$ for $m$ to be as follows.

\[
x \rightarrow y \rightarrow z \rightarrow \text{rest of the processes}
\]

That is, the timely paths to all processes lead from $x$ to $y$ then to $z$. If $\mathcal{A}$ is such that $x$ sends $m$ to $y$, then, by assumption, $y$ does not forward $m$ to $z$. Therefore, no process except for $y$ gets $m$ through timely packets. By definition of the timely message, $m$ is not timely received by these processes. If $x$ does not send $m$ to $y$, then none of the processes receive a timely message. In either case, contrary to the initial assumption, $\mathcal{A}$ does not timely deliver $m$ to all processes in the network. \hfill \square

The below corollary follows from Lemma 1.

**Corollary 2** It requires $\Omega(n^2)$ packets to timely deliver a message in the general propagation model.

Combining Corollary 2 and Theorem 5 we obtain Corollary 3.

**Corollary 3** In the general propagation model, there does not exist a message and packet efficient implementation of Omega.

**Proposition 1** There exists a message efficient implementation of Omega in the general propagation model where each correct process can send reliable messages to the leader.

The algorithm that proves the above proposition is a straightforward extension of the second algorithm in Aguilera et al. [2] where every process re-sends received messages to every outgoing channel.

### 4.3 Dependable Channel Model

Unlike the general propagation model where dependability properties of a channel cannot be established, if timely and reliable channels are allowed, the packet and
message efficient implementation of Omega is possible. However, the super packet efficiency implementation is not.

**Lemma 2** In any message efficient implementation of Omega, each correct process must have a fair-lossy path to the leader.

**Proof** Assume there is a message-efficient implementation $A$ of Omega where there is a correct process $x$ that does not have a fair-lossy path to the leader. According to Corollary 1, $x$ itself may not be elected the leader. Assume there is a computation $\sigma_1$ of $A$ where process $y \neq x$ is elected the leader. Note that fair-lossy channels are not type discriminating. That is, if $x$ does not have a fair-lossy path to $y$, but has a fair lossy path to some other process $z$, then $z$ does not have a fair-lossy path to $y$ either. Thus, there must be a set of processes $S \subseteq N$ such that $x \in S$ and $y \notin S$ that do not have fair-lossy paths to processes outside $S$.

Since $A$ is message efficient, processes of $S$ only send a finite number of messages to $y$. Consider another computation $\sigma_2$ that shares its prefix with $\sigma_2$ up to the point were the last message from processes of $S$ is received outside of $S$. After that, all messages from $y$ to processes in $S$ and all messages from $S$ to processes outside of $S$ are lost. That is in $\sigma_2$, $y$ does not have timely, or every fair-lossy, paths to processes of $S$. It is possible that some other process $w$ is capable of timely communication to all processes in the network. However, since $A$ is efficient, no other process but $y$ is supposed to send infinitely many messages.

Since all messages from $S$ are lost, $\sigma_1$ and $\sigma_2$ are indistinguishable for the correct processes outside of $S$. Therefore, they elect $y$ as the leader. However, processes in $S$ receive no messages from $y$. Therefore, they have to elect some other process $u$ to be the leader. This means that $A$ allows correct processes to output different leaders. That is, $A$ is not an implementation of Omega.

We define a **source** to be a process that does not have incoming timely channels.

**Lemma 3** To timely deliver a message in the dependable channel model, each recipient needs to send it across every outgoing channel to a source, except for possibly the channels leading to the origin and the sender.

**Proof** Assume the opposite. There exists an algorithm $A$ in the dependable channel model that timely delivers message $m_1$ from origin $x$ to all processes in the network such that some process $y$ that timely receives $m_1$ does not forward it to the source process $z$. Consider the propagation graph $T_1$ for $m_1$ to be as follows.

$$x \rightarrow \{\text{all processes but } z\} \cup y \rightarrow z$$

That is, the only timely path for message $m_1$ to source $z$ is from $x$ to $y$ to $z$. If $A$ is such that $x$ sends $m_1$ to $y$, then, by assumption, $y$ does not forward $m_1$ to $z$, then $z$ does not timely receive the message.

Now, since $z$ is a source, the propagation graph $T_2$ for message $m_2$ may be different than $T_1$. Let the propagation graph $T_2$ for $m_2$ be as follows.

$$x \rightarrow \{\text{all processes but } z\} \cup w \rightarrow z$$
The only timely path to process $z$ is via $x$ to $w$ to $z$. This time, assume $y$ forwards $m_2$ to $z$ but $w$ does not. Thus, $z$ does not timely receive $m_2$.

Therefore, for source $z$ to timely receive all messages, each recipient must forward all message to $z$.

Observe that Lemma 3 states that the timely delivery of a packet requires $n$ messages per source. If the number of sources is proportional to the number of processes in the network, we obtain the following corollary.

**Corollary 4** It requires $\Omega(n^2)$ packets to timely deliver a message in the dependable channel model where the number of sources is proportional to $n$.

**Theorem 6** In the dependable channel model, the following conditions are necessary and sufficient for the existence of a packet and message efficient implementation of Omega: (i) there is at least one process $l$ that has an eventually timely path to every correct process, and (ii) every correct process has a fair-lossy path to $l$.

**Proof** We demonstrate sufficiency by presenting, in the next section, an algorithm that implements Omega in the dependable channel model with the conditions of the theorem.

We now focus on proving necessity. Let us address the first condition of the theorem. Assume there is a message and packet efficient implementation $\mathcal{A}$ of Omega in the dependable channel model even though no process has eventually timely paths to every correct process. Let there be a computation of $\mathcal{A}$ where some process $x$ is elected the leader even though $x$ does not have a timely path to each correct process.

According to Corollary 1, $x$ needs to send infinitely many timely messages. According to Corollary 4, each such message requires $\Omega(n^2)$ packets. That is, $\mathcal{A}$ may not be message and packet efficient. This proves the first condition of the theorem. The second condition immediately follows from Lemma 2.

The below theorem shows that (plain) efficiency is all that can be achieved with the necessary conditions of Theorem 6. That is, even if these conditions are satisfied, super packet efficiency is not possible.

**Theorem 7** There does not exist a message and super packet efficient implementation of Omega in the dependable communication model even if there is a process $l$ with an eventually timely path to all correct processes and every correct process has fair-lossy paths to $l$.

**Proof** Assume the opposite. Suppose there exists a super packet efficient algorithm $\mathcal{A}$ that implements Omega in the network where some process $l$ has an eventually timely path to all correct processes and every correct process has fair-lossy paths to $l$.

Without loss of generality, assume the number of processes in the network is even. Divide the processes into two sets $S_1$ and $S_2$ such that the cardinality of both sets is $n/2$. Refer to Fig. 1 for an illustration. $S_1$ is completely connected by timely channels. Similarly, $S_2$ is also completely connected by timely channels. The dependability of channels between $S_1$ and $S_2$ is immaterial at this point.
Consider a computation $\sigma_1$ of $\mathcal{A}$ on this network where all processes in $S_1$ are correct and all processes in $S_2$ crashed in the beginning of the computation. Since $\mathcal{A}$ is an implementation of Omega, one process $l_1 \in S_1$ is elected the leader. Since $\mathcal{A}$ is message efficient, only $l_1$ sends messages infinitely often. Since $\mathcal{A}$ is super packet efficient, only $O(n)$ channels carry these messages infinitely often. Since the network is completely connected, there are $(n/2)^2$ channels leading from $S_1$ to $S_2$. This is in $O(n^2)$. Thus, there is at least one channel $(x, y)$ such that $x \in S_1$ and $y \in S_2$ that does not carry messages from $l_1$ infinitely often.

Let us consider a computation $\sigma_2$ of $\mathcal{A}$ where all processes in $S_2$ are correct and all processes in $S_1$ crash in the beginning of the computation. Similar to $\sigma_1$, there is a process $l_2 \in S_2$ that is elected the leader in $\sigma_2$, and there is a channel $(w, z)$ such that $w \in S_2$ and $z \in S_1$ that carries only finitely many messages of $l_2$.

We construct a computation $\sigma_3$ of $\mathcal{A}$ as follows. All processes are correct. Channel dependability inside $S_1$ and $S_2$ is as described above. All channels between $S_1$ and $S_2$ are completely lossy, i.e., they lose every transmitted message. An exception is channel $(x, y)$ that becomes timely as soon as it loses the last message it is supposed to transmit. Similarly, channel $(w, z)$ becomes reliable as soon as it loses the last message.

To construct $\sigma_3$, we interleave the actions of $\sigma_1$ and $\sigma_2$ in an arbitrary manner. Observe that to processes in $S_1$ computations $\sigma_1$ and $\sigma_3$ are indistinguishable. Similarly, to processes in $S_2$, the computations $\sigma_2$ and $\sigma_3$ are indistinguishable.

Let us examine the constructed computation closely. Sets $S_1$ and $S_2$ are completely connected by timely channels, and $(x, y)$, connecting $S_1$ and $S_2$ is eventually timely. This means that $l_1$ has an eventually timely path to every correct process in the network. Moreover, due to channel $(w, z)$, every process has a fair-lossy path to $l_1$. That is, the conditions of the theorem are satisfied. However, the processes of $S_1$ elect $l_1$ as their leader while the processes of $S_2$ elect $l_2$. This means that the processes do not agree on the single leader. That is, contrary to the initial assumption, $\mathcal{A}$ is not an implementation of Omega. The theorem follows.

5 **M\textit{EPO}: Message and Packet Efficient Implementation of Omega**

In this section we present an algorithm we call M\textit{EPO} that implements Omega in the fair-lossy channel communication model. As per Theorem 6, we assume that there is at least one process that has an eventually timely path to every correct process in the network and every correct process has a fair-lossy path to this process.
constants
    p // process identifier
    N // set of network process identifiers, cardinality is n
    timers[p] length is TO, i.e. initially an arbitrary integer

variables
    leader, initially ∅ // local leader
    phases[n], initially zero // current phase number
    edges[n][n], initially zero // edge fault weights
    arbs[n], initially arbitrary // arborescences
    timers[n], initially timers[p] on, others off

    length of timers[ν] : ν ≠ p is arbitrary // timer to send/receive a message
    shout, initially zero // process id to send alive to all neighbors

actions
    timeout(timers[q]) →
        if p = q then // own/sender timeout
            // compute arb rooted in p based on edges
            newArb = arborescence(edges, p)
            newLeader := minWeight(arbs[r] : r ≠ p : on(timers[r])), newArb
        if leader ≠ newLeader then // leadership changes
            if newLeader = p then // p gains leadership
                arbs[p] := newArb
                send startPhase(p, phases[p], arbs[p]) to N/p
            if leader = p then // p loses leadership
                phases[p] := phases[p] + 1
                send stopPhase(p, phases[p]) to N/p
                leader := newLeader
        else // leadership persists
            if leader = p then
                shout := shout + 1 mod n
                if shout ≠ p then
                    send alive(p, phases[p], shout) to arbs[p][p.children]
                    else // my turn to shout
                        send alive(p, phases[p], shout) to N/p
                    reset(timers[p]) // own timer never off
            // neighbor timeout/receiver timeout, assume failed, increase, do not reset
            else
                send failed(q, p, arbs[q](p.parent)) to N/p
                increase(timers[q])

receive startPhase(q, phase, arb) for the first time →
    // if new phase, propagate message, reset timer
    if p ≠ q ∧ phase[q] ≤ phase then
        arbs[q] := arb
        phases[q] := phase
        send startPhase(q, phase, arb) to N/p
        reset(timers[q])

receive stopPhase(q, phase) for the first time →
    if p ≠ q ∧ phase[q] < phase then
        phases[q] := phase
        send stopPhase(q, phase) to N/p
        stop(timers[q])

receive alive(q, phase, sh) for the first time from r →
    if p ≠ q ∧ phase[q] = phase then
        if r = arbs[q](p.parent) then // received through arborescence
            if sh ≠ p then
                send alive(q, phase, sh) to arbs[q][p.children]
                else // my turn to shout
                    send alive(q, phase, sh) to N/p
                    reset(timers[q])
                else // received from elsewhere
                    if off(timers[q]) then
                        reset(timers[q])
                    receive failed(q, r, s) for the first time →
                        if p = q then // if p’s alive failed
                            edges[r][r] := edges[r][r] + 1 // increase weight of edge from parent
                        else
                            send failed(q, r, s) to N/p

 Springer
5.1 Algorithm Outline

The code of the algorithm is shown in Fig. 2. The starting point of \textit{MPO} is the algorithm by Aguilera et al. [2]: the reader who read their paper will find our algorithm familiar. The main idea of \textit{MPO} is for processes to attempt to claim the leadership of the network while discovering the reliability of its channels. Each process weighs each channel by the number of messages that fail to come across it. The lighter channel is considered more reliable. If a process determines that it has the lightest paths to all processes in the network, the process tries to claim leadership of the network.

The leadership is obtained in phases. First, the leader candidate sends a \textit{startPhase} message. Then, the candidate periodically sends an \textit{alive} message. In case an \textit{alive} fails to reach one of the processes on time, the recipient replies with \textit{failed}. The size of \textit{startPhase} depends on the network size. The size of the other message types is constant.

The routes of the messages vary. Messages that are only sent finitely many times are \textit{broadcast}: sent across every channel in the network. Once one process receives such a message for the first time, the process re-sends it along all of its outgoing channels. Specifically, \textit{startPhase}, \textit{stopPhase} and \textit{failed} are broadcast. The leader sends \textit{alive} infinitely often. Hence, for the algorithm to be packet efficient, \textit{alive} has to be sent only along selected channels. Message \textit{alive} is routed through the channels that the origin believes to be the most reliable.

Specifically, \textit{alive} is routed along the channels of a minimum weight \textit{arborescence}: a directed tree rooted in the origin reaching all other processes. The arborescence is computed by the origin once it claims leadership. It is sent in the \textit{startPhase} that starts a phase. Once each process receives the arborescence, the process stores it in the \textit{arbs} array element for the corresponding origin. After receiving \textit{alive} from a particular origin, the recipient consults the respective arborescence and forwards the message to the channels stated there.

In addition to routing \textit{alive} along the arborescence, each process takes turns sending the leader’s \textit{alive} to all its neighbors. The reason for this is rather subtle: see Theorem 7 for details. Due to crashes and message losses, \textit{arbs} for the leader at various processes may not reach every correct process. For example, it may lead to a crashed process. Thus, some processes may potentially not receive \textit{alive} and, therefore, not send \textit{failed}. Since \textit{failed} are not sent, the leader may not be able to distinguish such a state from a state with a correct \textit{arbs}.

To ensure that every process receives \textit{alive}, each process, in turn, sends \textit{alive} to its every neighbor rather than along most reliable channels. Since only a single process sends to all neighbors a particular \textit{alive} message, the packet complexity remains $O(n)$.

Message \textit{failed} is sent if a process does not receive a timely \textit{alive}. This message carries the parent of the process that was supposed to send the \textit{alive}. That is, the sender of \textit{failed} blames the immediate ancestor in the arborescence. Once the origin of the missing \textit{alive} receives \textit{failed}, it increments the weight of the appropriate edge in \textit{edges} that stores the weights of all channels. If a process has timely outgoing paths to all processes in the network, its arborescence in \textit{edges} convergences to these paths.
5.2 Action Specifics

The algorithm is organized in five actions. The first is a timeout action, the other four are message-receipt actions.

The timeout action handles two types of timers: sender and receiver. Process $p$’s own timer ($q = p$) is a sender timer. It is rather involved. This timer is always on since the process resets it after processing. First, the process computes the minimum weight of the arborescence for each leader candidate. A process is considered a leader candidate if its timer is on. Note that since $p$’s own timer is always on, it is always considered.

The process with the minimum weight arborescence is the new leader. If the leadership changes ($\text{leader} \neq \text{newLeader}$), further selection is made. If $p$ gains leadership ($\text{newLeader} = p$), then $p$ starts a new phase by updating its own minimum-weight arborescence and broadcasting $\text{startPhase}$. If $p$ loses leadership, it increments its phase and broadcasts $\text{stopPhase}$ bearing the new phase number.

If the leadership persists ($\text{leader} = \text{newLeader}$) and $p$ is the leader, it sends $\text{alive}$. Process $p$ keeps track of whose turn it is to send $\text{alive}$ to all its neighbors in the $\text{shout}$ variable. The variable’s value rotates among the ids of all processes in the network.

The neighbor timer ($q \neq p$) is a receiver timer. If the process does not get $\text{alive}$ on time from $q$, then $p$ sends $\text{failed}$. In case the process sends $\text{failed}$, it also increases the timeout value for the timer of $q$ thus attempting to estimate the channel delay.

For our algorithm, the timer integers are as follows. The sender timer is an arbitrary constant integer value $\text{TO}$. This value controls how often $\text{alive}$ is sent. It does not affect the correctness of the algorithm. Receiver timers initially hold an arbitrary value. The timer integer is increased every time there is a timeout. Thus, for an eventually timely channel, the process is able to estimate the propagation delay and set the timer integer large enough that the timeout does not occur. For non-timely channels, the timeout value may increase without bound.

The next four actions are message receipt handling. Note that a single process may receive packets carrying the same message multiple times across different paths. However, every process handles the message at most once: when it encounters it for the first time. Later duplicate packets are discarded.

The second action is $\text{startPhase}$ handling. The process copies the arborescence and phase carried by the message, rebroadcasts it and then resets the $\text{alive}$ receiver timer associated with the origin process. The third action is the receipt of $\text{stopPhase}$, which causes the recipient to stop the appropriate timer.

The fourth action is $\text{alive}$ handling. If $\text{alive}$ is the matching phase, it is further considered. If $\text{alive}$ comes through the origin’s arborescence, the receiver sends $\text{alive}$ to its children in the origin’s arborescence or broadcasts it. The process then resets the timer to wait for the next $\text{alive}$. If $\text{alive}$ comes from elsewhere, that is, it was the sender’s turn to send $\text{alive}$ to all its neighbors, then $p$ just resets the timeout and waits for an $\text{alive}$ to arrive from the proper channel. This forces the process to send $\text{failed}$ if $\text{alive}$ does not arrive from the channel of the arborescence.

The last action is $\text{failed}$ handling. If $\text{failed}$ is in response to an $\text{alive}$ originated by this process ($p = q$) then the origin process increments the weight of the edge
from the parent of the reporting process to the process itself according to the message arborescence. If failed is not destined to this process, \( p \) rebroadcasts it.

### 5.3 Correctness Proof Definitions

Throughout this section, \( l \) is the identifier of the process that has eventually timely paths to all other processes. For simplicity, we assume that \( l \) is the single such process. Denote \( B \) as the maximum number of steps in any timely channel propagation delay. Process \( p \) is a local leader if \( \text{leader}_p = p \), i.e., the process elected itself the leader. A process may be a local leader but not the global leader. That is, several processes may be local leaders in the same state. Let \( \text{realArbs}(x) \) for the origin process \( x \) be the relation defined by \( \text{arbs}[x](y.\text{children}) \) at every process \( y \). That is, \( \text{realArbs}(x) \) is the distributed relation that determines how alive messages are routed if they are originated by \( x \).

**Lemma 4** For any local leader process \( x \) and another correct process \( y \) such that \( y \) is not reachable from \( x \) through timely channels over correct processes in \( \text{realArbs}(x) \), either (i) \( \text{realArbs}(x) \) changes or (ii) \( x \) loses leadership, changes phase or receives infinitely many failed messages.

**Proof** To prove the lemma, it is sufficient to show that if \( \text{realArbs}(x) \) does not change and \( x \) does not lose the leadership or change phase, then \( x \) receives infinitely many failed.

Let \( S \) be a set of correct processes that are reachable from \( x \) through timely channels and through correct processes in \( \text{realArbs}(x) \). Since \( y \) is not reachable from \( x \), \( S \neq N \). Recall that every process has fair-lossy paths to all processes in the network. Therefore, there is such a path from \( x \) to \( y \). This means that there is a process \( z \in S \) such that it has a fair-lossy channel to \( w/\in S \).

Let us examine process \( w \) closer. The network is completely connected. Therefore, all other processes from \( S \) have channels to \( w \). Note that at least one channel, from \( z \) is fair-lossy. Moreover, since \( w \) does not belong to \( S \), if \( \text{realArbs}(x) \) reaches \( w \), the path to \( w \) is not timely.

Since \( x \) is a local leader and does not lose its leadership, it sends infinitely many alive messages. Other processes forward these alive messages along \( \text{realArbs}(x) \). Also, by the design of the algorithm, every process takes turn sending alive to all of its neighbors rather than forwarding it along \( \text{realArbs}(x) \). Let us examine the receipt of these messages by \( w \).

Process \( z \) belongs to \( S \). That is, the path from \( x \) to \( z \) in \( \text{realArbs}(x) \) is timely. This means that it receives and sends infinitely many alive messages originated by \( x \). Since the channel from \( z \) to \( w \) is fair-lossy, infinitely many of these alive messages are delivered to \( w \). In addition, \( w \) possibly receives alive from other processes of \( S \). Since, none of these channels are part of \( \text{realArbs}(x) \), when \( w \) receives alive from processes in \( S \), it resets the corresponding receive timer only when the timer is off. The timer is turned off only when the timeout is executed and failed is broadcast.

The only possible way this receive timer is reset without the timeout action execution is when \( w \) receives alive through \( \text{realArbs}(x) \). However, the path from \( x \) to \( w \) in
realArbs(x) is not timely. By the definition of non-timely paths, there are infinitely many computation segments of arbitrary fixed length where no alive from x is delivered to w. This means that, regardless of the timeout variable value at w, the alive messages generate receiver timeouts. That is, infinitely many timeouts are executed at w. Each timeout generates a failed message broadcast by w. Since there are infinitely many broadcasts, infinitely many succeed in reaching x. Hence, the lemma.

Lemma 5 If each process $x \neq l$ is a local leader in infinitely many states then it receives infinitely many failed messages.

Proof Let $x \neq l$ be a local leader in infinitely many states of a particular computation of the algorithm. Once a process assumes local leadership, it may lose it either by (i) increasing the weight of its minimum weight arborescence or (ii) by recording an arborescence $arbs[y]$ for a process y with lower weight than $arbs[x]$.

A process increases the weight of its arborescence only when it gets a failed message. Thus, to prove the lemma we need to consider the second case only.

Since $x$ is a local leader in infinitely many states, it must gain local leadership back after losing it to another process $y$. By the design of the algorithm, the weight of the arborescence of any process in $arbs$ may only increase. This means that once $x$ gains the leadership back from $y$, $x$ may not lose it to $y$ again without increasing the weight of its own minimum weight arborescence. Thus, either $x$ increases the weight of its arborescence or, eventually, it has the lightest arborescence among the leader candidates.

In case $x$ has the lightest arborescence, it either becomes heavier than some other leader candidate’s arborescence or $x$ gets infinitely many failed. However, only the latter part of the statement needs to be proven since $x$ gains leadership infinitely often.

If $x$ is a local leader, it does not send startPhase or stopPhase. Let us consider the state where all startPhase packets are delivered. In this case realArbs(x) does not change. Since $x \neq l$, even if all correct processes are reachable from x in realArbs(x), some links in realArbs(x) are not timely. Then, according to Lemma 4, x gets infinitely many failed.

To summarize, if $x \neq l$ is a local leader in infinitely many states, it receives infinitely many failed.

Lemma 6 Process l is a local leader in infinitely many states.

Proof According to Lemma 5 either each process $x \neq l$ stops gaining local leadership or the weight of its minimum arborescence grows infinitely high. If the latter is the case, $x$ has to gain and lose local leadership infinitely often. In this case, it sends startPhase infinitely many times. Message startPhase is broadcast. Since every process $x$ has fair-lossy paths to $l$, by the definition of fair-lossy paths, infinitely many broadcasts succeed. This means that the weight of $arbs[x]$ at $l$ grows without bound. Therefore, if $l$ loses local leadership, it gains it back infinitely often.

The below lemma follows immediately from the operation of the algorithm. Recall that $B$ is the maximum number of steps between sending and receiving a packet in a timely channel.
Lemma 7 The timer length of timers\([l]\) at every process either stops increasing or it reaches \(TO + B \ast (n - 1)\).

And the below lemma follows from the assumption that the leader has an eventually timely path to every correct process.

Lemma 8 In every computation, there is a suffix where each broadcast message sent by \(l\) is timely delivered to every correct process.

Lemma 9 An edge leading to process \(x\) in a timely path in \(realArbs(l)\) generates only finitely many failed.

Proof The origin starts every phase with \(startPhase\), then periodically sends zero or more \(alive\) and then possibly ends the phase with a \(stopPhase\) that carries the phase number greater than that in \(alive\) and \(startPhase\).

Message \(failed\) is generated only when the timer expires at the receiving process. The timer is reset by \(startPhase\) and \(alive\). The timer is stopped by \(stopPhase\).

We prove the lemma by showing that the timer reset by messages of a particular phase expires only finitely many times. We start our consideration from the point of the computation where the conditions of Lemmas 7 and 8 hold.

Only \(alive\) and \(startPhase\) may reset the timeout. Since the conditions of Lemma 8 hold, \(startPhase\) is delivered within \(B(n - 1)\) computation steps to all processes. Message \(alive\) may be received earlier than \(startPhase\). However, since such \(alive\) carries a phase number that differs from the number stored at the recipient process, the message is ignored. If \(alive\) arrives after \(startPhase\), the reasoning is similar to the case where \(alive\) is sent after \(startPhase\), which is to be considered next.

Every \(alive\) sent after \(startPhase\) delivery travels over the timely path in \(realArbs(l)\). At most every \(TO\) number of steps, either another \(alive\) or \(stopPhase\) is sent. Since the path in \(realArbs(l)\) is timely, \(alive\) arrives at most after \(TO + B(n - 1)\) steps. Due to Lemma 8, the same is true of \(stopPhase\). That is, after \(alive\) is received, either another \(alive\) or \(stopPhase\) is received within \(TO + B(n - 1)\) steps. The receipt of \(alive\) resets the timeout. The receipt of \(stopPhase\) stops it. Due to Lemma 7, the timer does not expire.

Moreover, after the receipt of \(stopPhase\), the subsequent \(alive\) messages are ignored since \(stopPhase\) carries a greater phase number. That is, after \(stopPhase\) is received, the timer is never reset or expires due to the messages of this phase.

Lemma 10 Every non-timely edge in \(realArbs(l)\) leading to a correct process either gets removed or \(l\) gets infinitely many failed messages.

Proof Due to Lemma 6, process \(l\) is a local leader in infinitely many states. Through the argument similar to that of Lemma 5, we can show that eventually either \(l\) gets \(failed\) and increases the weight of its minimum arborescence or its minimum arborescence becomes the lightest among the leader candidates. Then, \(l\) can lose leadership only if it gets \(failed\).
In this case, according to Lemma 4, \( l \) receives infinitely many failed messages or either loses leadership, changes phase or changes \( realArbs(l) \). Observe that \( l \) may change phase only when it receives \( failed \). It loses leadership only if it gets \( failed \). The change of \( realArbs(l) \) happens only when \( l \) broadcasts \( startPhase \) after changing phase and, therefore, getting \( failed \). Due to Lemma 6, it gains the leadership back infinitely often.

That is, in any case, as long as \( realArbs(l) \) contains a non-timely edge leading to a correct process, \( l \) gets infinitely many \( failed \).

The below lemma follows from Lemmas 9 and 10.

**Lemma 11** Every computation of \( MPO \) contains a suffix where each channel of \( realArbs(l) \) is timely.

**Lemma 12** Every computation of \( MPO \) contains a suffix where \( realArbs(l) \) is the same as \( arbs[l] \) in process \( l \).

**Proof** We start our consideration from the point where the conditions of Lemma 11 hold. Suppose \( realArbs(l) \) and \( arbs[l] \) differ for some process \( x \). By the design of the algorithm, this may happen only if \( arbs[l] \) in process \( x \) has an earlier phase than in \( l \). However, since phases differ, \( alive \) sent by \( l \) are ignored by \( x \). This leads to either \( x \) sending \( fail \) to \( l \) or claiming leadership. In either case, \( l \) sends \( startPhase \). According to Lemma 8, this broadcasts succeeds, which synchronizes \( arbs[l] \) and \( realArbs(l) \).

**Theorem 8** Algorithm \( MPO \) is a message and packet efficient implementation of \( Omega \) in the fair-lossy channel model.

**Proof** First, we prove that \( MPO \) implements \( Omega \). Indeed, Lemma 6 shows that \( l \) is a local leader in infinitely many states. Lemmas 9 and 10 show that \( l \) gets finitely many \( failed \). According to Lemma 5, every process \( x \neq l \) either stops being a local leader or gets infinitely many \( failed \). This means that at any process the arborescence of \( l \) is eventually lighter than any other leader contender.

According to Lemma 6, \( l \) sends infinitely many \( alive \) messages along \( realArbs(l) \). Due to Lemma 10, \( realArbs(l) \) eventually has no non-timely channels. Since \( l \), according to Lemma 10, receives only finitely many \( failed \), due to Lemma 4, \( realArbs(l) \) eventually has timely paths from \( l \) to every correct process. According to Lemma 12, \( realArbs(l) \) and \( arbs[l] \) are eventually the same.

This means that \( l \) is a leader contender at every correct process. Since it has the lightest arborescence, it becomes the leader at every correct process. In other words, \( MPO \) is a correct implementation of \( Omega \).

By the design of the algorithm, once \( l \) has the lightest arborescence and all correct processes drop out of leadership contention, \( l \) is the only process that sends \( alive \) messages. By definition, \( MPO \) is message efficient.

The messages are routed along \( arbs[l] \). It is an arborescence. Hence, the number of such messages is in \( O(n) \). In addition, each process takes a turn sending \( alive \) to its
neighbors. This is another $O(n)$ packets. Therefore, the packet complexity of $MPO$ is in $O(n)$.

6 Conclusion

6.1 Summary

In summary, a multi-hop implementation of the Omega failure detector is more likely to succeed in electing and maintaining a leader than a single-hop implementation. Although we prove that a super packet efficient implementation of Omega is not possible, by enhancing the asynchronous system model with necessary timeliness and reliability assumptions, we prove correct our packet efficient $MPO$ implementation.

6.2 $MPO$ Extensions

The algorithm trivially works in a non-completely connected network provided that the rest of the assumptions used in the algorithm design, such as eventually timely paths from the leader to all correct processes, are satisfied. Similarly, the algorithm works correctly if the channel reliability and timeliness is origin-related. That is, a channel may be timely for some, not necessarily incident, process $x$, but not for another process $y$. Algorithm $MPO$ may be modified to use only constant-size messages. The only non-constant size message is $startPhase$. However, the message type is supposed to be timely. So, instead of sending a single large message, the modified $MPO$ may instead send a sequence of fixed-size messages with the content to be re-assembled by the receivers. If one of the constituent messages does not arrive on time, the whole large message is considered lost.

6.3 Future Research

With the number of available Omega implementations, it would be interesting to carry out a performance comparison as to which particular failure detector features are of greater practical benefit. Another avenue of practical verification is to estimate the channel timeliness of the practical consensus systems and how it affects the performance of our failure detector.

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, pp. 328–337. ACM (2004)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega in systems with weak reliability and synchrony assumptions. Distrib. Comput. 21(4), 285–314 (2008)
3. Anta, A.F., Raynal, M.: From an asynchronous intermittent rotating star to an eventual leader. IEEE Trans. Parallel Distrib. Syst 21(9), 1290–1303 (2010)
4. Arantes, L., Greve, F., Sens, P., Simon, V.: Eventual leader election in evolving mobile networks. In: International Conference on Principles of Distributed Systems (OPODIS), vol. 8304, pp. 23–37 (2013)
5. Biely, M., Widder, J.: Optimal message-driven implementations of omega with mute processes. ACM Trans. Auto. Adaptive Syst. 4(1), 4 (2009)
6. Bollobás, B.: Random graphs, 2 edn. Cambridge University Press, Cambridge (2001). http://www.worldcat.org/isbn/0521797225
7. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
8. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
9. Chandy, K.M., Misra, J.: Parallel program design: A Foundation. Addison-Wesley, Boston (1988)
10. Charron-Bost, B., Függer, M., Nowak, T.: Approximate consensus in highly dynamic networks. arXiv:1408.0620 (2014)
11. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. In: Symposium on Self-Stabilizing Systems, pp. 219–233. Springer (2007)
12. Delporte-Gallet, C., Devismes, S., Fauconnier, H., Larrea, M.: Algorithms for extracting timeliness graphs. In: Structural Information and Communication Complexity, 17th International Colloquium, SIROCCO 2010, Sirince, Turkey, June 7-11, 2010. Proceedings, pp. 127–141 (2010)
13. Erdős, P.: Rānyi: On random graphs I. Publ. Math. Debrecen 6, 290–297 (1959)
14. Fernández, A., Jiménez, E., Raynal, M.: Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In: International Conference on Dependable Systems and Networks (DSN’06), pp. 166–178. IEEE (2006)
15. Fernández-Campusano, C., Larrea, M., Cortiñas, R., Raynal, M.: A communication-efficient leader election algorithm in partially synchronous systems prone to crash-recovery and omission failures. In: Proceedings of the 17th International Conference on Distributed Computing and Networking, p. 8. ACM (2016)
16. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985). https://doi.org/10.1145/3149.214121
17. Gilbert, E.N.: Random graphs. Ann. Math. Statist. 30(4), 1141–1144 (1959). https://doi.org/10.1214/aoms/1177706098
18. Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Chasing the weakest system model for implementing ω and consensus. IEEE Trans. Dependable Sec. Comput. 6(4), 269–281 (2009)
19. Jiménez, E., Arévalo, S., Fernández, A.: Implementing unreliable failure detectors with unknown membership. Inf. Process. Lett. 100(2), 60–63 (2006)
20. Lafuente, A., Larrea, M., Soraluze, I., Cortiñas, R.: Communication-optimal eventually perfect failure detection in partially synchronous systems. J. Comput. Syst. Sci. 81(2), 383–397 (2015)
21. Larrea, M., Fernández, A., Arévalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Trans. Comput. 53(7), 815–828 (2004)
22. Malkhi, D., Oprea, F., Zhou, L.: Ω meets paxos: Leader election and stability without eventual timely links. In: Distributed Computing, pp. 199–213. Springer (2005)
23. Mostefaoui, A., Mouragaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 351–351. IEEE Computer Society (2003)
24. Mostefaoui, A., Raynal, M., Travers, C.: Time-free and timer-based assumptions can be combined to obtain eventual leadership. IEEE Trans. Parallel Distrib. Syst. 17(7), 656–666 (2006)
25. Peleg, D.: Distributed computing: a locality-sensitive approach. SIAM (2000)
26. Prisco, R.D., Lampson, B.W., Lynch, N.A.: Revisiting the PAXOS algorithm. Theor. Comput. Sci. 243(1-2), 35–91 (2000)