An Eventually Perfect Failure Detector on ADD Channels Using Clustering

Laine Rumreich and Paolo A. G. Sivilotti

The Ohio State University, Columbus OH, USA
rumreich.1@osu.edu, paolo@cse.ohio-state.edu

Abstract. We present an implementation of an eventually perfect failure detector, $\Diamond$P, in a partitionable network with arbitrary topology. This network is built on weak assumptions using ADD channels, which are defined as requiring the existence of constants K and D, not known to the processes, such that for every K consecutive messages sent, at least one is delivered within time D. The best previous implementation of $\Diamond$P on ADD channels uses a heartbeat-based approach with time-to-live values for messages. The message size complexity of this existing implementation is $\mathrm{O}(En \log n)$ for the entire network for any given heartbeat. In contrast, the solution presented in this paper organizes the network into clusters, each with a single leader, to reduce the message size complexity to $\mathrm{O}(En)$. The algorithm is structured as a series of superpositioned layers and a proof of correctness is given for the $\Diamond$P oracle based on these layers. We compare the performance of the cluster-based failure detector with that of the best previous solution on various topologies using simulation.

Keywords: failure detector · ADD channel · superpositioning · clustering

1 Introduction

1.1 Context

A failure detector is a distributed oracle that can be queried for information about crashed processes. Failure detectors are a type of consensus problem in that all nodes in a network must agree on the status of all other nodes. However, consensus problems cannot be solved deterministically in an asynchronous system with failures [7, 5]. This is known as the impossibility result. Chandra et al. [3] proposed using unreliable failure detectors as an alternative to circumvent this impossibility result. Unreliable oracles make mistakes by incorrectly suspecting correct processes or failing to suspect crashed processes. Although these oracles are allowed to make mistakes, this paper concerns an eventually perfect failure detector, meaning the oracle is only allowed to make these mistakes for a finite amount of time.

The eventually perfect failure detector class, $\Diamond$P, was defined by Chandra and Toueg [3] and is characterized by the properties of *strong completeness* and *eventual strong accuracy*. These properties are defined as the following:

1. Strong completeness: every crashed process is eventually permanently suspected by every process
2. Eventual strong accuracy: Every correct process is eventually not suspected by every correct process

Many previous implementations of ◊P assume models of partial synchrony with bounded message delay or reliable communication [6, 14]. Sastry and Pike [15] proposed an algorithm using much weaker assumptions for the communication channel. This communication channel is known as an Average Delayed/Dropped (ADD) channel. This channel promises a certain number of messages are received in a given amount of time. Both the number of dropped messages and the amount of time are unknown to the process, however.

The failure detector defined in this paper is built on top of an ADD channel because it is the weakest communication channel used in previous implementations of ◊P. Kumar and Welch [10] built upon the initial failure detector result on ADD channels by allowing an arbitrariy connected network. This implementation also concerns a ◊P algorithm on a network of arbitrary topology. Finally, Vargas, Rajsbaum, and Raynal constructed an eventually perfect failure detector on ADD channels for an arbitrarily connected network with an improved message size complexity [17]. This improvement was based on the addition of a time-to-live value and heartbeat-based approach, both of which are common techniques in networking.

1.2 Motivation

The result from Vargas, Rajsbaum, and Raynal using time-to-live values with a heartbeat-based algorithm has a message complexity size of O($nlogn$) per node per heartbeat, which is equivalent to O($Enlogn$) per heartbeat. This complexity is improved to O(En) in this implementation using a hierarchy of clusters in the network. This hierarchy of clusters can be constructed dynamically from an arbitrary network.

1.3 Contribution

The central contribution of this work is in the reduction in complexity size of the previous best implementation of an eventually perfect failure detector built on ADD channels. This reduction in size complexity requires the additional logical complexity of clustering, a hierarchy of nodes, and the use of superpositioning techniques to organize and structure the algorithm. A proof of correctness for this algorithm is constructed based on the correctness of the underlying algorithm and the addition of an overlay network of nodes to transmit information between leaders. An additional contribution of this work is the addition of a simulation of the algorithm to show convergence for the failure detector comparing the cluster-based algorithm discussed in this work to the original implementation. This simulation compares a variety of topologies and shows that networks with a topology optimized for clusters have improved convergence performance.

2 Background

2.1 Eventually Perfect Failure Detectors

$\Diamond$P can give unreliable information about process crashes for only a finite prefix. Eventually, it must provide *only* correct information about crashed processes. The oracle $\Diamond$P is of interest because it is both powerful and implementable. $\Diamond$P is sufficiently powerful to solve many fundamental problems such as consensus [3]. Unlike Perfect (P) [3], Strong (S) [3], and Marabout detectors [8], $\Diamond$P is the only oracle implementable in partially synchronous systems [12].

2.2 ADD Channels

The communication channel this work is built on is known as an Average Delayed/Dropped (ADD) channel [15]. An ADD channel from nodes p to q, given unknown constants K and D, satisfies the following two properties:

- The channel does not create or duplicate messages
- For every K consecutive messages sent by p to q, at least one is delivered to q within D time

In addition, processes on ADD channels can fail only by crashing, pairs of processes are connected via two reciprocal ADD channels, and each process has access to a local clock that generates ticks at a constant rate.

2.3 Heartbeat and Time-to-Live

Both heartbeats and time-to-live (TTL) values are used in this work based on the logic from the Vargas et al. [17] version of the algorithm for a failure detector on ADD channels. In that algorithm, heartbeats are used to keep track of the distance of a message across the network. This heartbeat will eventually fade out when a process fails. TTL values are then used to track the intensity level of a message such that as messages travel through the network, the intensity decreases. TTLs are commonly used in networking to limit the lifetime of message packets [11].

2.4 Group Membership and Clustering

Group Membership is a similar but separate problem from Consensus. The value that must be agreed upon, group membership, is allowed to change due to asynchronous failures, and nonfaulty processes may be removed when they are erroneously suspected to have crashed. Techniques used to circumvent the impossibility of Consensus can also be applied to solve the Group Membership Problem [2]. Group membership can be used to generate clusters based on various properties. An unreliable failure detector can be used to generate these groups. Many algorithms can be used to form clusters statically [9].

2.5 Superpositioning

Superpositioning is a general technique for structuring complex algorithms by decomposing these algorithms into distinct layers [1]. In the context of action systems, as used in this paper, each layer augments the layers below with new features (actions and variables) while preserving the functional properties of those lower layers [4]. In order for this preservation of functional correctness to hold, however, the superpositioning of actions and variables must be done in a disciplined way. In particular, while actions introduced in higher layers can *use* (read) the values of variables from lower layers, they must not *modify* (write) these variables. Any variables introduced in a layer, on the other hand, can be both read and written by actions within that layer. This approach allows for a separation of concerns. Each layer can make use of the services provided by lower layers in order to implement some new functionality. That functionality, in turn, is available to higher layers that respect the discipline of superpositioning.

3 Algorithm

3.1 Overview

The following algorithm solves $\Diamond$P using a two-level hierarchical version of the Vargas et al. [17] $\Diamond$P algorithm. In this new hierarchical version of the algorithm, the network is categorized into mutually exclusive clusters that form the first level of the hierarchy. Each cluster has a single leader and the set of leaders form the top level of the hierarchy. Each cluster in the network can then be conceptualized as a single node, and $\Diamond$P is solved between this collection of nodes. Additionally, $\Diamond$P is solved locally within clusters.

The clusters described for this algorithm must be constructed deliberately for the algorithm to work. The construction of these clusters is described here but not implemented in the pseudocode. The clusters must have the following characteristics: (1) Each cluster has a single leader (2) Each node is either a leader or a cluster node that is assigned to a single leader (3) The entire cluster must be connected using nodes only in the current cluster- that is, it remains connected if all other nodes are removed.

3.2 Description

Cluster nodes communicate information to other nodes in their cluster using a variable called *cluster_HB_bag*, which contains heartbeat and TTL information for their cluster only. These cluster nodes also *forward* information across an overlay network to and from their leader and their neighbors. Leader nodes participate in both the cluster-level and network-level failure detection. They consolidate failure information from their cluster and pass along heartbeat information about other clusters to the leaders. They also update their cluster on the rest of the network through the overlay network messages. In addition to the algorithms for failure detection, a *leader election* algorithm occurs when a leader

node has been suspected of failing by one of the nodes in its cluster. The leader election algorithm determines a new leader for a particular cluster. If a cluster has been partitioned, the leader election algorithm will result in one additional leader. The new leader information must also be propagated to all other nodes in the network.

3.3 Layers of Superpositioning

The algorithm presented in this section is constructed in layers rather than as one large algorithm. Some of these layers run on the same nodes, but they are presented in this way to organize the tasks for the heartbeat algorithm separated for the overlay network and clustering requirements. To prove that these layers do not negatively interfere with each other, superpositioning is used. To guarantee the safety properties of the lower levels, lower levels cannot read variables that are written at a higher level. Variables in lower levels of the algorithm also guard higher levels against performing tasks and can trigger actions to occur. For example, a cluster node that becomes separated from its assigned leader will trigger the leader election layer to run. Fig. 1 illustrates the five layers of the algorithm including where the shared variables originate and are used in higher layers.

Layer Name	Variables Defined	Variables Accessed
1. Heartbeat - Leader		cluster, cluster_suspect, leaders, leader_clusters
2. Forwarding Overlay		cluster, leader_clusters
3. Leader Notification	leaders, leader_clusters	cluster, leader
4. Leader Election	leader	cluster, cluster_suspect
5. Heartbeat - cluster	cluster, cluster_suspect	

Fig. 1: Algorithm Layering and Shared Variables

3.4 Layer 1: Heartbeat - Leader

The top layer of this algorithm, the heartbeat running on leaders, is the primary method of achieving $\Diamond$P. The overlay network running on the layer below allows this layer to be abstracted as running on leaders that are directly connected without cluster nodes in between. Information is sent and received by leaders using messages that are composed of sets of three-tuples. A single three-tuple includes heartbeat information about the leader plus a simple suspect list composed of only `true` and `false` values comprising the information about that leader's cluster. The set of three-tuples relays heartbeat information about the entire network. This algorithm updates the suspect list for leader nodes, which stores the status of every other node in the network and is thus the primary data structure in the proof of $\Diamond$P for leader nodes. Algorithm 1 contains the pseudocode for this layer.

Algorithm 1 Heartbeat - Leader **Leader Node** p

```
Constants:
 1: T, n, neighbors
Variables:
 2: leader_clusters, leaders
 3: leader_bag, cluster_suspect, cluster
 4: clock() ← 1
 5: for each i in Π do
 6:   suspect[i] ← false
 7: end for
 8: for each i in leaders do
 9:   leader_lastHB[i] = 0
10:   leader_timeout[i] = T
11:   leader_TTL[i] ← 1
12: end for

Information Send
13: every T units of time of clock()
14: begin:
15:   leader_bag ← {(p, |leaders| − 1, cluster_suspect)}
16:   for each i ∈ leaders \ {p} do
17:     if leader_TTL[i] > 1 then
18:       external_cluster_suspect ← GET_CLUSTER_SUSPECT(i, leader_clusters)
19:       leader_bag ← leader_bag ∪ {(i, leader_TTL[i] − 1, external_cluster_suspect)}
20:     end if
21:   end for

22:   for each q ∈ neighbors \ cluster do  ▷ send to non-cluster neighbors
23:     send(< leader_bag >) to q
24:   end for

25:   for each q ∈ neighbors ∩ cluster do  ▷ send to cluster neighbors
26:     is_outgoing ← true
27:     TLL ← |cluster| − 1
28:     send(< leader_bag, TTL, is_outgoing >) to q
29:   end for
30: end

Information Receive
31: upon receiving < lead_bag, TTL, is_outgoing > from q ∈ cluster
32: begin:
33:   if not(is_outgoing) then
34:     for each (r, m, array) ∈ lead_bag such that r ∉ neighbors \ {q} do
35:       if leader_TTL[r] ≤ m then
36:         leader_TTL[r] ← m
37:         for each node ∈ leader_clusters.get(r) do
38:           suspect[node] ← array[node]
39:         end for
40:         if suspect[r] = true then
41:           suspect[r] ← false
42:           ESTIMATE_TIMEOUT(r)
43:         end if
44:         leader_lastHB[r] ← clock()
45:       end if
46:     end for
47:   end if
48: end

49: upon receiving < lead_bag > from q ∉ cluster
50: begin:
51:   ▷ Begin from line 34
52: end

53: procedure ESTIMATE_TIMEOUT(r)
54:   if r ∈ leaders then
55:     leader_timeout[r] ← 2 · leader_timeout[r]
56:   end if
57: end procedure

58: procedure GET_CLUSTER_SUSPECT(r, leader_clusters)
59:   current_cluster ← leader_clusters.get(r)
60:   j ← 0
61:   for each i ∈ current_cluster do
62:     array[j] ← suspect[i]
63:     j ← j + 1
64:   end for
65:   return array
66: end procedure

Leader Timeout
66: when leader_timeout[q] = clock() − leader_lastHB[q]
67: begin:
68:   suspect[q] ← true
69: end
```

3.5 Layer 2: Overlay Network Message Forwarding

The next layer of this algorithm is the overlay network layer, which transfers information between leader nodes through the cluster nodes. This layer runs only on cluster nodes. The basic structure of the overlay network is to forward messages by broadcasting information to neighbors each heartbeat. Neighbors within the current node's cluster also receive information about whether the current message is *incoming* or *outgoing*. Outgoing messages are those that originated from the current node's leader and will thus be ignored by the leader. Cluster nodes will also update their local suspect list using information from outgoing messages if they are more recent than the current information. For cluster nodes, this suspect list is used to store the status information of every other node in the network and is thus the primary data structure in the proof of $\Diamond$P for cluster nodes. The detailed pseudocode for this algorithm is included in the Appendix.

Algorithm 2 Forwarding Overlay **Cluster Node** p

Constants:

1: $T, n, neighbors$

Variables:

2: $clock() \leftarrow 1$
3: $leader_clusters, cluster$
4: **for** each i in Π **do**
5: $suspect[i] \leftarrow false$ ▷ suspect list for entire network
6: **end for**
7: $FORWARD_MESSAGES(leader_clusters, cluster)$

3.6 Layer 3: Leader Notification

A cluster may become partitioned due to failures in the network. When this occurs, a new leader will be elected for one of the halves of the partitioned cluster. When a new leader is elected due to a partitioned cluster, this algorithm spreads that information to existing leader nodes. This allows the leader heartbeat algorithm to accurately update the information from leaders about their clusters. The detailed pseudocode for this algorithm is included in the Appendix.

Algorithm 3 Leader Notification **Cluster Node** p

Variables:

1: $clock() \leftarrow 1$
2: $leaders, leader_clusters, cluster, leader$
3: **if** leader changes **then**
4: $leaders, leader_clusters \leftarrow ALERT(cluster, leader)$
5: **end if**

3.7 Layer 4: Leader Election

The leader election layer of the algorithm performs a basic leader election for a single cluster when the leader fails or the cluster is partitioned. Both of these actions require the addition of a new leader in the algorithm. The pseudocode for the leader election algorithm on ADD channels has been constructed previously [16] and is not included here. The triggering of a new leader election action is guarded by information from the layer below.

Algorithm 4 Leader Election **Node** p

Constants:

1: $T, n, neighbors$

Variables:

2: $clock() \leftarrow 1$
3: $leader, cluster, cluster_suspect$
4: **if** $cluster_suspect[leader]$ **then**
5: $\quad leader \leftarrow \Omega(cluster)$
6: **end if**

3.8 Layer 5: Heartbeat - Cluster

The bottom layer of this algorithm is the heartbeat ◇P algorithm within clusters. It is running on every node and updates cluster-level suspect information. The logic of this layer is separated based on if the current node is a leader or not. Information from other clusters is not transferred in this layer. Cluster-level information that is transferred by leaders is updated in this level. Cluster-level suspect information is used to inform the shared `cluster` variable.

Algorithm 5 Heartbeat - Cluster: **Cluster Node** p

Constants:

1: $T, n, neighbors$

Variables:

2: $clock() \leftarrow 1, cluster, cluster_bag$
3: **for** each i in $cluster$ **do**
4: $\quad cluster_lastHB[i] = 0$
5: $\quad cluster_suspect[i] \leftarrow false$
6: $\quad cluster_timeout[i] = T$
7: $\quad cluster_TTL[i] \leftarrow 1$
8: **end for**

Cluster Send

9: **every** T units of time of $clock()$
10: **begin:**
11: $\quad cluster_bag \leftarrow \{(p, |cluster| - 1)\}$
12: $\quad$ **for** each $i \in cluster \setminus \{p\}$ **do**
13: $\quad\quad$ **if** $cluster_suspect[i] = false$ and $cluster_TTL[i] > 1$ **then**
14: $\quad\quad\quad cluster_bag \leftarrow cluster_bag \cup \{(i, cluster_TTL[i] - 1)\}$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: $\quad$ **for** each $q \in neighbors \cap cluster$ **do**
18: $\quad\quad$ **send**($< cluster_bag >$) to q
19: $\quad$ **end for**
20: **end**

Cluster Receive

21: **upon** receiving < gr_bag > from $q \in cluster$
22: **begin:**
23: **for** each $(r, m) \in gr_bag$ such that $r \notin neighbors \setminus \{q\}$ **do**
24: **if** $cluster_TTL[r] \leq m$ **then**
25: $cluster_TTL[r] \leftarrow m$
26: **if** $cluster_suspect[r] = true$ **then**
27: $cluster_suspect[r] \leftarrow false$
28: $ESTIMATE_TIMEOUT[r]$
29: **end if**
30: $cluster_lastHB[r] \leftarrow clock()$
31: **end if**
32: **end for**
33: **end**

34: **procedure** ESTIMATE_TIMEOUT(r)
35: $cluster_timeout[r] \leftarrow 2 \cdot cluster_timeout[r]$
36: **end procedure**

Timeout

36: **when** $cluster_timeout[q] = clock() - cluster_lastHB[q]$
37: **begin:**
38: $cluster_suspect[q] \leftarrow true$
39: **end**

Algorithm 6 Heartbeat - Cluster: **Leader Node** p

Constants:

1: $T, n, neighbors$

Variables:

2: $clock() \leftarrow 1$
3: $cluster, cluster_bag$
4: **for** each i in $cluster$ **do**
5: $cluster_lastHB[i] = 0$
6: $cluster_suspect[i] \leftarrow false$ ▷ suspect list for cluster only
7: $cluster_timeout[i] = T$
8: $cluster_TTL[i] \leftarrow 1$
9: **end for**

Cluster Send

10: **every** T units of time of $clock()$
11: **begin:**
12: $cluster_bag \leftarrow \{(p, |cluster| - 1)\}$
13: **for** each $i \in cluster \setminus \{p\}$ **do** ▷ only include the cluster members
14: **if** $cluster_suspect[i] = false$ and $TTL[i] > 1$ **then**
15: $cluster_bag \leftarrow cluster_bag \cup \{(i, TTL[i] - 1)\}$
16: **end if**
17: **end for**
18: **for** each $q \in neighbors \cap cluster$ **do** ▷ only send to neighbors in cluster
19: **send**(< $cluster_bag$ >) to q
20: **end for**
21: **end**

Cluster Receive

22: **upon** receiving < *gr_bag* > from $q \in$ *cluster*
23: **begin:** ▷ receive from a cluster member
24: **for** each $(r, m) \in$ *gr_bag* such that $r \notin$ *neighbors* $\setminus \{q\}$ **do**
25: **if** $TTL[r] \leq m$ **then**
26: $TTL[r] \leftarrow m$
27: **if** *cluster_suspect*$[r]$ = *true* **then**
28: *cluster_suspect*$[r] \leftarrow$ *false*
29: $ESTIMATE_TIMEOUT[r]$
30: **end if**
31: *cluster_lastHB*$[r] \leftarrow$ *clock*()
32: **end if**
33: **end for**
34: **end**

35: **procedure** ESTIMATE_TIMEOUT(r)
36: **if** $r \in$ *cluster* **then**
37: *cluster_timeout*$[r] \leftarrow 2 \cdot$ *cluster_timeout*$[r]$
38: **end if**
39: **end procedure**

Timeout

66: **when** *cluster_timeout*$[q]$ = *clock*() − *cluster_lastHB*$[q]$
67: **begin:**
68: *cluster_suspect*$[q] \leftarrow$ *true*
69: **end**

4 Overlay Network

4.1 The Network Graph $G(t^\epsilon)$

The proof of ◇P will be constructed on the arbitrary network graph G after some point in time after all failures have occurred. Graph G at some time t is defined as $G(t) = (correct(\alpha, t), E')$ with $E' = \{(u, v) | (u, v) \in E$ and $u, v \in correct(\alpha, t)\}$. Now define t^f to be the earliest time when all the failures in α have occurred. We must now show that after time t^f the graph G will no longer change, and all messages from failed processes cease to circulate.

Lemma 1. *The graph G has the following properties.*

1. $G(t) = G(t')$ *for all* $t, t' \geq t^f$
2. *There is a time* $t^\epsilon \geq t^f$ *after which the last message from the set of crashed processes is delivered to a correct process.*

Proof. Let Π be the set of all nodes in the network.
Part One: By construction, at time t^f all failures have occurred. Now it must be the case that $crashed(\alpha, t) = crashed(\alpha, t')$ for all $t, t' \geq t^f$. Since $correct(\alpha, t) = \Pi \setminus crashed(\alpha, t)$, then by the definition of $G(t)$ it is true that $G(t) = G(t')$ for all $t, t' \geq t^f$.

Part Two: The algorithm described in this work has two separate node types: leader nodes and cluster nodes. Every faulty leader clearly sends a finite number of heartbeat messages before crashing. Then, by the properties of the ADD channel, these messages are lost, delivered or experience arbitrary delays. Thus there exists a time $t^\epsilon \geq t^f$ after which the last message sent by the set of

faulty leader processes is delivered. Cluster nodes also send a finite number of heartbeat messages and the proof applies by the same argument. Cluster nodes also forward messages to and from other nodes. Each of these messages are bounded by a timeout value by construction and thus will stop being forwarded in some finite amount of time. Now it is clear that for both leader and cluster nodes, there exists a time $t^{\epsilon} \geq t^{f}$ after which the last message sent by the set of faulty leader processes is delivered.

From this point we will assume that we have reached time t^{ϵ} such that all failures have occurred and all messages from failed nodes have finished circulating through the network. Similarly, all references to graph G from this point on refer to some time after t^{ϵ}. https://www.overleaf.com/project/61dde39fc5e336dabd0727fe

The Overlay Network G' In this section we will now define a graph G'. In particular, we will show that G is an implementation of the model G'.

To define G', consider the model for a network defined by Vargas et al. [17] defining a set of processes $\Pi = \{1, 2, ..., n\}$ connected by ADD channels in both directions and represented an undirected graph $G' = (V, E)$ where $V = \Pi$. This network consists of *channels* and *processes*.

Channels The channels in this model are ADD channels, meaning they satisfy the definition from [15].

Processes Processes may fail only by crashing. Given an execution α, a process p is said to crash at time t if p does not perform any event inα after time t and crashed(α, t) is the set of all processes that have crashed by time t. A process p is correct at time t if p has not crashed by time t and $correct(\alpha, t) = \Pi \setminus crashed(\alpha, t)$.

We will define a relation from the overlay network in the algorithm to the underlying network. The following section will demonstrate how the overlay network can be viewed as an implementation of the underlying network.

4.2 Overlay Network Relation

Consider the topology of the network defined in the $\Diamond$P with clusters algorithm in this paper. We will show that this graph topology can be considered an implementation of the underlying network defined using ADD channels.

Recall that the network is partitioned into distinct, connected clusters, each with a single leader. Consider each cluster as a single node with the same connections to other clusters as the original topology, but as a single connection. A representation of the connection from the network to the abstract view of the network is shown in Fig. 2 and Fig. 3. The following sections will prove that this abstraction results in a network with ADD channel properties.

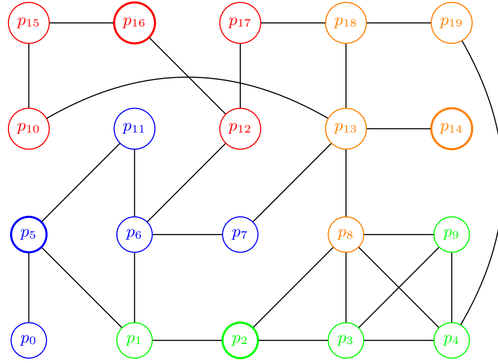

Fig. 2: Network Topology Overlay, G

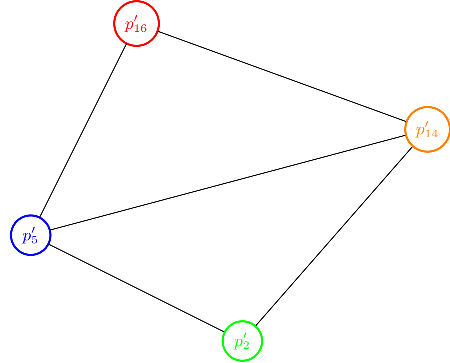

Fig. 3: Network Topology, G'

5 Proof of ◊P

5.1 Proof Outline

The proof of ◊P for this algorithm is based on two parts:

1. A proof that this algorithm is an implementation of the abstraction constructed by Vargas et al. [17], represented by the graph G'
2. The proof of ◊P by Vargas et al. [17] on graph G'

The rest of this section concerns part one only. The proof of ◊P follows from part one and Vargas et al. [17].

5.2 Proof of Overlay Network Model

This section concerns G', which is a graph of edges and nodes. The proof that the overlay network forms an abstraction between the network of leaders and the underlying network requires proving two things: that the edges of G' still satisfy the properties of ADD channels and that the nodes of G' still operate and fail in the correct way. In particular, this proof concerns the graph G' after some time t^{ϵ}.

Edges - ADD Channel An ADD channel from p to q has associated constants K, D, such that the following properties are satisfied:

1. The channel does not create, modify, or duplicate messages
2. For every K consecutive messages sent by p to q, at least one is delivered to q within time D

Property 1 The first property of the overlay network that the implementation must satisfy is that the channel does not create, modify, or duplicate messages. The channels in the network topology will be modeled to not modify or duplicate messages, even though the underlying implementation does not fit these requirements. The overlay network modifies incoming messages by adding a TTL, but this value is removed when the message is relayed to the cluster leader, so we can model the network as one that does not modify the messages. The overlay network also allows duplicate messages because any new incoming message from outside or within the cluster will be broadcast to every neighbor in the cluster. However, these messages are labeled with a unique identifier, so if duplicates arrive at the leader, they can be ignored, and the model preserves the no-duplicates property.

Property 2 We must also show that for every K consecutive messages sent by p to q, at least one is delivered to q within time D. Without loss of generality, we will consider messages from leader 1, ℓ_1, to leader 2, ℓ_2. Let the path from ℓ_1 to ℓ_2 be $\ell_1, p_1, p_2, ..., p_n, \ell_2$. Now, each p is a node in an overlay network or is a cluster, which can be abstracted as a single leader node. Since each channel along this path is an ADD channel, there exists some K_i, D_i for each i along the path from ℓ_1 to ℓ_2. Now, a message from ℓ_1 is guaranteed to be received by ℓ_2 by the $K^t h$ consecutive message where K is: $K = \Pi_{i=0}^{n} K_i$. Similarly, this message will be delivered within time D where D is: $D = \Pi_{i=0}^{n} D_i$.

Nodes - Cluster Model To match the underlying network, each cluster must be modeled by a single node. Clusters are running both a ◊P algorithm and a transmission network to transfer information. The overlay network must correctly transfer information from the edge of the cluster to the leader and back. The path must also be limited to cluster members only. The TTL of these messages guarantees that they will not survive past the length of the longest path of unique nodes in the cluster. Messages are broadcasted to and from nodes, so they are guaranteed to reach the leader and then back to the edge nodes. Incoming and outgoing messages are separated so that information coming from outside the cluster does not leave the cluster without modification. The primary requirement to show is that leaders have correct information that can then be sent through the overlay network to other leaders. The proof of correctness for the underlying ◊P algorithm applies directly to within-cluster communication. Finally, failures that cause clusters to no longer be connected or no longer have a functioning leader result in the construction of new clusters. These new clusters must be connected and have a leader, thus the model of the overlay network always remains consistent with the underlying network.

5.3 ◊ P

The Vargas et al. [17] proof of correctness for ◊P now applies to the network G' at time t^ϵ after all failures have occurred. Now, since G is an implementation of the G' abstract model, the proof of ◊P for the arbitrary network running the algorithm is complete.

6 Complexity

The size complexity of this algorithm is calculated based on the size and number of messages sent in the network for a single heartbeat. The following equation represents that value, where E represents the number of edges in the graph, E' is the number of edges within clusters, n is the number of nodes, and ℓ is the number of leaders:

$$O(E' \cdot \frac{n}{\ell} \log n) + O(E(n + \ell \cdot \log n)) \tag{1}$$

These two terms are separated based on the complexity at both hierarchy levels. The first term is the complexity of the heartbeat algorithm within clusters. The second term is the complexity of the leader-level heartbeat algorithm including the overlay network.

1. Cluster Heartbeat: The sum over each cluster node of the degree of the node times the (worst case) size of its `HB_bag`. The size of this variable is a $(\log n)$ encoding times the size of the cluster, sent to every neighbor.
2. Leader Heartbeat and Forwarding: A message including the `HB_lead_bag` (size number of leaders times $\log n$) plus an array of size n with the status (1 or 0) of every node in the network, sent across every edge in the network two times.

In order to minimize the complexity, the leader size must be chosen. At the extremes, which includes having each node as a leader and having no leaders, the complexity of the algorithm with clusters does not improve from $\mathrm{O}(E \cdot n \log n)$. The complexity of the algorithm with clusters is reduced to $\mathrm{O}(En)$ when the number of leaders is chosen to be $\log n$, $\sqrt{n}$ or any number of other values.

6.1 Leader Election Complexity

An additional aspect of the complexity of this algorithm is the leader election that occurs upon the failure of a leader node. The complexity of this algorithm would be a maximum of $\mathrm{O}(E \cdot \log n)$ [13]. Thus this additional cost does not impact the overall message size complexity of this algorithm.

7 Experimental Results

In order to measure the expected-case behavior of our cluster-based $\Diamond$P algorithm, a simulation was conducted over a variety of network topologies and failure modes [18]. Since the message space complexity can be calculated analytically (see previous section), this simulation focussed on the relative delay in convergence to an accurate suspect list in $\Diamond$P. The links between nodes are assumed to be ADD channels with identical delay and drop characteristics, so the convergence time is reported in number of heartbeats. Topologies considered range from a single chain at one extreme to a fully connected graph on the other.

Table 1 compares the convergence times for the original and cluster-based algorithms on a variety of topologies. Each topology has 100 nodes. Three different scenarios were tested with each topology: one node suspecting one other node, every node suspecting one node, and every node suspecting every other node.

Table 1: Heartbeats Until Convergence

Network	Leaders	Average Degree	**Original Algorithm** 1-1/M-1/M-M	**Cluster Algorithm** 1-1/M-1/M-M
Chain	5	.99	223/223/295	98/98/122
Chain	10	.99	223/223/295	50/50/62
Chain	15	.99	223/223/295	38/38/44
Chain Conn. Clusters	5	3.09	1/4/4	3/8/8
Chain Conn. Clusters	10	1.54	1/4/4	3/8/8
Chain Conn. Clusters	15	1.27	1/4/4	3/8/8
Fully Conn. Clusters	5	42	4/4/4	6/6/6
Fully Conn. Clusters	10	9.5	4/4/4	6/6/6
Fully Conn. Clusters	15	8.5	4/4/4	6/6/6
Fully Conn. Leaders	5	4.5	1/4/4	3/5/14
Fully Conn. Leaders	10	3.25	1/4/4	3/17/29
Fully Conn. Leaders	15	3.07	1/4/4	3/5/44
Average Connectedness	5	3.2	1/4/4	3/5/14
Average Connectedness	10	3.1	1/4/4	3/17/29
Average Connectedness	15	3.07	1/4/4	3/5/44
Fully Connected	5	49.5	1/1/3	3/5/5
Fully Connected	10	49.5	1/1/3	3/5/5
Fully Connected	15	49.5	1/1/3	3/5/5

The time to convergence is similar for both algorithms. The cluster-based algorithm performs better for sparsely connected graphs with large diameter. For densely connected graphs, the original algorithm's advantage diminishes with the number of erroneous suspicions.

8 Conclusion

This paper demonstrates a modification and improvement upon a previous implementation of an eventually perfect failure detector on ADD channels based on a hierarchical, cluster-based overlay network and superpositioning. This algorithm demonstrates a reduction in the message size complexity of the best previous implementation of $\Diamond$ P of complexity $\mathrm{O}(E \cdot n \log n)$ down to complexity $\mathrm{O}(En)$. Additionally, a simulation demonstrates a similar time to convergence for the cluster-based algorithm compared to the previous implementation. This convergence time was tested using a variety of network topologies. Future work could be done to expand the hierarchy of this algorithm to more than two levels to further improve the message complexity size.

Appendix

Algorithm 7 Transmit Leader Information **Node** p

```
1: procedure ALERT
2:   every T units of time of clock()
3:   begin:
4:       ▷ Where ADD_Transmit is the link level reliable transmission of information
5:     ADD_TRANSMIT(cluster, leader)
6:   end
7: end procedure
```

Algorithm 8 Transmit Messages **Cluster Node** p

```
1: procedure FORWARD_MESSAGES
2:   upon receiving < leader_bag, TTL, is_outgoing > from q ∈ cluster
3:   begin:
4:     local_leader_bag ← leader_bag
5:     UPDATE_SUSPECT(leader_bag)
6:   end

7:   every T units of time of clock()
8:   begin:
9:     ▷ Send to other clusters
10:    if is_outgoing then ▷ send outgoing information to non-cluster neighbors
11:      for each q ∈ neighbors \ cluster do
12:        send(< local_leader_bag >) to q
13:      end for
14:    end if
15:  end
16:  ▷ Within-cluster Send/Receive
17:  every T units of time of clock()
18:  begin:
19:    if TTL > 1 then ▷ Transfer incoming and outgoing information to other cluster members
20:      for each q ∈ neighbors ∩ cluster do ▷ only send to neighbors in cluster
21:        send(< leader_bag, TTL − 1, is_outgoing >) to q
22:      end for
23:    end if
24:  end

25:  ▷ Receive from other clusters
26:  upon receiving < leader_bag > from q ∉ cluster
27:  begin:
28:    for each q ∈ neighbors ∩ cluster do
29:      is_outgoing ← false
30:      TTL ← |cluster| − 1
31:      message ← leader_bag, TTL, is_outgoing
32:    end for
33:  end

34:  every T units of time of clock()
35:  begin:
36:    for each q ∈ neighbors ∩ cluster do
37:      send(< message >) to q
38:    end for
39:  end
40: end procedure
```

```
41: ▷ Create local copy of suspect list
42: procedure UPDATE_SUSPECT(leader_bag)
43:   for each (r, m, array) ∈ leader_bag do
44:     if leader_TTL[r] ≤ m then
45:       leader_TTL[r] ← m
46:       for each node ∈ leader_clusters.get(r) do
47:         suspect[node] ← array[node]
48:       end for
49:       if suspect[r] = true then
50:         suspect[r] ← false
51:         ESTIMATE_TIMEOUT(r)
52:       end if
53:       leader_lastHB[r] ← clock()
54:     end if
55:   end for
56: end procedure
```

References

1. Back, R.J., Sere, K.: Superposition refinement of parallel algorithms. In: FORTE (1991)
2. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (jul 1996). https://doi.org/10.1145/234533.234549
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (mar 1996). https://doi.org/10.1145/226643.226647
4. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley Publishing Company, Reading, Massachusetts (1988)
5. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. J. ACM **34**(1), 77–97 (jan 1987). https://doi.org/10.1145/7531.7533
6. Fetzer, C., Schmid, U., Susskraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05). pp. 271–280 (2005). https://doi.org/10.1109/ICDCS.2005.57
7. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (apr 1985). https://doi.org/10.1145/3149.214121
8. Guerraoui, R.: On the hardness of failure-sensitive agreement problems. INFORMATION PROCESSING LETTERS **79**(2), 99–104 (2001)
9. Jain, A.K., Dubes, R.C.: Algorithms for Clustering Data. Prentice-Hall, Inc., USA (1988)
10. Kumar, S., Welch, J.: Implementing $\Diamond P$ with bounded messages on a network of add channels. Parallel Processing Letters **29** (08 2017). https://doi.org/10.1142/S0129626419500026
11. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach. Pearson, Boston, MA, 7 edn. (2016)
12. Larrea, M., Fernandez, A., Arevalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Transactions on Computers **53**(7), 815–828 (2004). https://doi.org/10.1109/TC.2004.33
13. Rajsbaum, S., Raynal, M., Vargas, K.: Brief announcement: Leader election in the add communication model. In: Stabilization, Safety, and Security of Distributed Systems: 22nd International Symposium. p. 229–234 (2020). https://doi.org/10.1007/978-3-030-64348-5_18
14. Raynal, M., Mourgaya, E., Mostefaoui, A.: Asynchronous implementation of failure detectors. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). p. 351. IEEE Computer Society, Los Alamitos, CA, USA (jun 2003). https://doi.org/10.1109/DSN.2003.1209946
15. Sastry, S., Pike, S.: Eventually perfect failure detectors using add channels. In: ISPA. pp. 483–496 (08 2007)
16. Sergio Rajsbaum, M.R., Godoy, K.V.: Leader election in arbitrarily connected networks with process crashes and weak channel reliability. 22nd International Symposium on Stabilization, Safety, and Security of Distributed Systems (2020)
17. Vargas, K., Rajsbaum, S., Raynal, M.: An eventually perfect failure detector for networks of arbitrary topology connected with add channels using time-to-live values. Parallel Processing Letters **30**(02), 2050006 (2020). https://doi.org/10.1142/S0129626420500061

18. Zirger, K., Rumreich, L., Sivilotti, P.: Failure Detector with Clusters Simulation. https://github.com/osu-rsrg (2022)