Algorithms For Extracting Timeliness Graphs*

Carole Delporte-Gallet
University Paris Diderot
Carole.Delporte@liafa.jussieu.fr

Stéphane Devismes
University Joseph Fourier (Grenoble)
Stephane.Devismes@imag.fr

Hugues Fauconnier
University Paris Diderot
Hugues.Fauconnier@liafa.jussieu.fr

Mikel Larrea
University of the Basque Country
Mikel.Larrea@ehu.es

Abstract

We consider asynchronous message-passing systems in which some links are timely and processes may crash. Each run defines a *timeliness graph* among correct processes: (p, q) is an edge of the timeliness graph if the link from p to q is timely (that is, there is bound on communication delays from p to q). The main goal of this paper is to approximate this timeliness graph by graphs having some properties (such as being trees, rings,...). Given a family S of graphs, for runs such that the timeliness graph contains at least one graph in S then using an *extraction algorithm*, each correct process has to converge to the same graph in S that is, in a precise sense, an approximation of the timeliness graph of the run. For example, if the timeliness graph contains a ring, then using an extraction algorithm, all correct processes eventually converge to the same ring and in this ring all nodes will be correct processes and all links will be timely.

We first present a general extraction algorithm and then a more specific extraction algorithm that is communication efficient (*i.e.*, eventually all the messages of the extraction algorithm use only links of the extracted graph).

1 Introduction

Designing fault-tolerant protocols for asynchronous systems is highly desirable but also highly complex. Some classical agreement problems such as *consensus* and *reliable broadcast* are well-known tools for solving more sophisticated tasks in faulty environments (e.g., [17, 15]). Roughly speaking, with consensus processes must reach a common decision on their inputs, and with reliable broadcast processes must deliver the same set of messages.

It is well known that consensus cannot be solved in asynchronous systems with failures [14], and several mechanisms were introduced to circumvent this impossibility result: *randomization* [7], *partial synchrony* [11, 12] and *(unreliable) failure detectors* [6].

Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about the process crashes. Each process can access a local failure detector module that monitors the processes of the system and maintains a list of processes that are suspected of having crashed.

Several classes of failure detectors have been introduced, *e.g.*, $\mathcal{P}$, $\mathcal{S}$, Ω, etc. Failure detectors classes can be compared by reduction algorithms, so for any given problem P, a natural question is "*What is the weakest failure detector (class) that can solve P ?*". This question has been extensively studied for several problems in systems *with infinite process memory* (*e.g.*, uniform and non-uniform versions of consensus [5, 13],

*This work has been supported in part by the ANR projet *SHAMAN*.

non-blocking atomic commit [9], uniform reliable broadcast [1, 19], implementing an atomic register in a message-passing system [9], mutual exclusion [10], boosting obstruction-freedom [16], set consensus [21, 22], etc.). This question, however, has not been as extensively studied in the context of systems *with finite process memory*.

In this paper, we consider systems where processes have finite memory, processes can crash and links can lose messages (more precisely, links are fair lossy and FIFO[1]). Such environments can be found in many systems, for example in sensor networks, sensors are typically equipped with small memories, they can crash when their batteries run out, and they can experience message losses if they use wireless communication.

In such systems, we consider (the uniform versions of) reliable broadcast, consensus and repeated consensus. Our contribution is threefold: First, we establish that the weakest failure detector for reliable broadcast is $\mathcal{P}^-$ — a failure detector that is almost as powerful than the perfect failure detector $\mathcal{P}$.[2] Next, we show that consensus can be solved using failure detector $\mathcal{S}$. Finally, we prove that $\mathcal{P}^-$ is the weakest failure detector for repeated consensus. Since $\mathcal{S}$ is strictly weaker than $\mathcal{P}^-$, in some precise sense these results imply that, in the systems that we consider here, consensus is easier to solve than reliable broadcast, and reliable broadcast is as difficult to solve as repeated consensus.

The above results are somewhat surprising because, when processes have infinite memory, reliable broadcast is easier to solve than consensus[3], and repeated consensus is not more difficult to solve than consensus.

Roadmap. The rest of the paper is organized as follows: In the next section, we present the model considered in this paper. In Section **??**, we show that in case of process memory limitation and possibility of crashes, $\mathcal{P}^-$ is necessary and sufficient to solve reliable broadcast. In Section **??**, we show that consensus can be solved using a failire detector of type $\mathcal{S}$ in our systems. In Section **??**, we show that $\mathcal{P}^-$ is necessary and sufficient to solve repeated consensus in this context.

For space considerations, all the proofs are relegated to an optional appendix.

2 Informal Model

Graphs. We begin with some definitions and notations concerning graphs. For a directed graph $G = \langle N, E\rangle$, $Node(G)$ and $Edge(G)$ denote N and E, respectively. Given a graph G and a set $M \subseteq Node(G)$, $G[M]$ is the *subgraph* of G induced by M, *i.e.*, $G[M]$ is the graph $\langle M, Edge(G)[M]\rangle$ where $(p, q) \in Edge(G)[M]$ if and only if $p, q \in M$ and $(p, q) \in Edge(G)$.

The tuple (X, Y) is a *directed cut* (*dicut* for short) of G if and only if X and Y define a partition of $Node(G)$ and there is no directed edge $(y, x) \in Edge(G)$ such that $x \in X$ and $y \in Y$. We say that G' is a *dicut reduction* from G if there exists a dicut (X, Y) of G such that $G' = G[X]$. A set S of graphs is *dicut-closed* if and only if it is closed under dicut reduction, namely if $G \in S$ then all the graphs obtained by a dicut-reduction of G are in S.

Processes and Links. We consider distributed systems composed of n processes which communicate by message-passing through directed links. We denote the set of processes by $\Pi = \{p_1, ..., p_n\}$. We assume that the communication graph is complete, *i.e.*, for each pair of distinct processes (p, q), there is a directed link from p to q.

[1] The FIFO assumption is necessary because, from the results in [20], if lossy links are not FIFO, reliable broadcast requires unbounded message headers.

[2] Note that $\mathcal{P} \subseteq \mathcal{P}^-$ and $\mathcal{P}^-$ is *unrealistic* according to the definition in [8].

[3] With infinite memory and fair lossy links, (uniform) reliable broadcast can be solved using Θ [4], and Θ is strictly weaker than (Σ, Ω) which is necessary to solve consensus.

A process may fail by crashing, in which case it definitively stops its local algorithm. A process that never crashes is said to be *correct*, *faulty* otherwise.

The (directed) links are *reliable*, *i.e.* every message sent through a link (p, q) is eventually received by q if q is correct and if a message m from p is received by q, m is received by q at most once, and only if p previously sent m to q.

The links being reliable, an implementation of the *reliable broadcast* [18] is possible. A reliable broadcast is defined with two primitives: `rbroadcast`$\langle m \rangle$ and `rdeliver`$\langle m \rangle$. Informally, after a correct process p invokes `rbroadcast`$\langle m \rangle$, all correct processes eventually `rdeliver`$\langle m \rangle$; after a faulty process p invokes `rbroadcast`$\langle m \rangle$, either all correct processes eventually `rdeliver`$\langle m \rangle$ or correct processes never `rdeliver`$\langle m \rangle$.

Timeliness. To simplify the presentation, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

We assume that every correct process p is *timely*, *i.e.*, there is a lower and an upper bound on the execution rate of p. Correct processes also have clocks that are not necessarily synchronized but we assume that they can accurately measure intervals of time.

A link (p, q) is *timely* if there is an unknown bound δ such that no message sent by p to q at time t may be received by q after time $t + \delta$.

A *timeliness graph* is simply a directed graph whose set of nodes are a subset of Π. The timeliness graph represents the timeliness properties of the links. Intuitively, for timeliness graph G, $Node(G)$ is the set of correct processes and (p, q) is in $Edge(G)$ if and only if the link (p, q) is timely.

Runs. An algorithm $\mathcal{A}$ consists of n deterministic (infinite) automata, one for each process; the automaton for process p is denoted $\mathcal{A}(p)$. The execution of an algorithm $\mathcal{A}$ proceeds as a sequence of process *steps*. Each process performs its steps atomically. During a step, a process may send and/or receive some messages and changes its state.

A run r of algorithm $\mathcal{A}$ is a tuple $r = \langle T, I, E, S \rangle$ where T is a timeliness graph, I is the initial state of the processes in Π, E is an infinite sequence of steps of $\mathcal{A}$, and S is a list of increasing time values indicating when each step in E occurred. A run must satisfy usual properties concerning sending and receiving messages. Moreover, we assume that (1) all correct processes make an infinite number of steps: $p \in Node(G)$ if and only if p makes an infinite number of steps in E and (2) the timeliness of links is deduced from the timeliness graph: $(p, q) \in Edge(G)$ if and only if the link (p, q) is timely in E.

In the following for run $r = \langle T, I, E, S \rangle$, $T(r)$ denotes T the timeliness graph of r, and $Correct(r)$ is the set of correct processes for the run r, namely, $Correct(r) = Node(T(r))$. Note that by definition, (p, q) is a timely link if and only if $(p, q) \in Edge(T)$.

Remark that in the definition given here a link may be timely even if no message is sent on the link. If link (p, q) is FIFO (*i.e.*, messages from p to q are received in the order they are sent) and p regularly sends messages to q, then the timeliness of these messages implies the timeliness of the link itself. So in the following we always assume that links are FIFO.

2.1 Some Systems

We say that timeliness graph G is *compatible with timeliness graph* G' if and only if (1) $Node(G) = Node(G')$ and (2) $Edge(G) \subseteq Edge(G')$. By extension, timeliness graph G is *compatible with run* r if G is compatible with $T(r)$, the timeliness graph of r. Hence, timeliness graph G is compatible with run r if $Node(G)$ is the set of correct processes in r and if (p, q) is an edge of G then (p, q) is timely in r.

A *system* $\mathcal{X}$ is defined as a set of timeliness graphs. The set of runs of system $\mathcal{X}$ denoted $R(\mathcal{X})$ is the set of all runs r such that there exists a timeliness graph G in $\mathcal{X}$ compatible with r.

Below, we define the systems considered in this paper:

- $\mathcal{ASYNC}$ is the set of all timeliness graphs G such that $Edge(G) = \emptyset$. In $\mathcal{ASYNC}$ there is no timeliness assumption about links and $R(\mathcal{ASYNC})$ is the set of all runs in an asynchronous system.
- $\mathcal{COMPLETE}$ is the set of all complete graphs whose nodes are the subsets of Π.
- $\mathcal{STAR}$ is the set of all timeliness graphs with a *source, i.e.,* $G \in \mathcal{STAR}$ if and only if $Node(G) \subseteq \Pi$ and there exists $p_0 \in Node(G)$ (the center of the star or the source) such that $Edge(G) = \{(p_0, q) | q \in Node(G) \setminus \{p_0\}\}$. Clearly a run r is in $R(\mathcal{STAR})$ if and only if there is at least one *source* in r.
- $\mathcal{TREE}$ is the set of all timeliness graphs G that are rooted directed trees, *i.e.,* $|Edge(G)| = |Node(G)| - 1$ and there exists p_0 in $Node(G)$ such that $\forall q \in Node(G)$, there is a directed path of G from p_0 to q. Clearly a run r is in $R(\mathcal{TREE})$ if and only if there is at least one timely path from a correct process to all correct processes.
- $\mathcal{RING}$ is the set of all timeliness graphs G such that G is a directed cycle (a ring). Clearly a run r is in $R(\mathcal{RING})$ if and only if there is a timely (directed) cycle over all correct processes.
- $\mathcal{SC}$ is the set of all timeliness graphs that are strongly connected. Clearly, a run r is in $R(\mathcal{SC})$ if and only if there exists a (directed) timely path between each pair of distinct correct processes.
- $\mathcal{BIC}$ is the set of all timeliness graphs G such that for all p, $q \in Node(G)$, there exist at least two distinct paths from p to q. $\mathcal{BIC}$ corresponds to the set of 2-strongly-connected graphs. Clearly, a run r is in $R(\mathcal{BIC})$ if and only if there exists at least two distinct timely paths between each pair of distinct correct processes.
- $\mathcal{PAIR}$ is the set of all timeliness graphs G such that $Edge(G) = \{(p_0, p_1), (p_1, p_0)\}$ with $p_0, p_1 \in Node(G)$ and $p_1 \neq p_0$. Clearly, a run r is in $R(\mathcal{PAIR})$ if and only if there exists two distinct correct processes p_0 and p_1 such that (p_0, p_1) and (p_1, p_0) are timely links.

3 Extraction Algorithms

Given a system $\mathcal{X}$, the goal of an *extraction algorithm* is to ensure that in each run r in $\mathcal{X}$, all correct processes eventually agree on the same element of $\mathcal{X}$ and that this element is, in some precise sense, an approximation of the timeliness graph of run r.

For example, in $\mathcal{RING}$, all processes have to eventually agree on some ring and this ring has to be compatible with the timeliness graph of the run. In particular this ring contains all the correct processes. However, the compatibility relation may be too strong: In many systems, it is not possible to distinguish between a crashed process and a correct one, so the graph G on which the processes eventually agree may contain crashed processes and then the graph is not exactly compatible with the run. Then we weaken the compatibility and impose only that the subgraph of G induced by the set of correct processes of the run is a dicut reduction of the timeliness graph of the run.

We now formally define what an extraction algorithm is. First, in such an algorithm, every process p maintains a local variable G_p which contains a timeliness graph. Then, we say that an algorithm *extracts a timeliness graph in* $\mathcal{X}$ if and only if for every run r in $\mathcal{X}$ there is a timeliness graph G (called the *extracted graph*) such that:

- *Convergence:* for all correct processes p there is a time t after which $G_p = G$

- *Compatibility:* $G[Correct(r)]$ is compatible with $T(r)$
- *Closure:* $G[Correct(r)]$ is a dicut reduction of G or is equal to G
- *Validity:* G is in $\mathcal{X}$

Remark that for all systems that contain $\mathcal{ASYNC}$ there is a trivial extraction algorithm: for each run processes extract the graph G such that $Node(G) = \Pi$ and $Edge(G) = \emptyset$.

A more constrained version of the extraction problem is the following: an algorithm $\mathcal{A}$ *extracts exactly* timeliness graphs in $\mathcal{X}$ if for every run r in system $\mathcal{X}$, the extracted graph G is compatible with $T(r)$. In this case, all correct processes eventually know the exact set of correct processes: it is the set of nodes of the extracted graph.

Some Results about Extraction Algorithms. First we show that an extraction algorithm may help to route messages using only timely links:

Lemma 3.1 *Let G be a graph extracted from run r, if (p, q) is in $Edge(G)$ and q is a correct process then p is correct.*

Proof. By contradiction, assume that p is not correct, then $(Correct(r), Node(G) - Correct(r))$ is not a dicut because $(p, q) \in Edge(G)$, $p \in Node(G) - Correct(r)$ and $q \in Correct(r)$, which contradicts the Closure property. □

From this lemma and the Compatibility property, we deduce directly:

Proposition 3.2 *If $(p = p_0, \ldots, p_i, \ldots, q = p_m)$ is a path in the extracted graph and p and q are correct processes, then for every i such that $0 \leq i < m$ the link (p_i, p_{i+1}) is timely and process p_i is correct.*

From a practical point of view, this proposition shows that the extracted graph may be used to route messages between processes using only timely links: the route from p to q is a path in the extracted graph (if any). All intermediate nodes are correct processes and agree on the extracted graph and then on the path.

For example with $\mathcal{TREE}$, the tree extracted by the algorithm enables to route messages from the root of the tree to any other processes and the routing uses only timely links.

Generally, the main goal of the extraction algorithm is not only to extract a graph G in $\mathcal{X}$ but also to ensure that $G[Correct(r)]$ is in $\mathcal{X}$ (even if the processes do not know the set of correct processes). In particular, this property is ensured if $\mathcal{X}$ is dicut-closed: the Closure property implies that $G[Correct(r)]$ is in $\mathcal{X}$.

Among the systems we consider, only system $\mathcal{PAIR}$ is not dicut-closed: $H = \langle \{x\}, \emptyset \rangle$ is a dicut reduction of $G = \langle \{x, y, z\}, \{(y, z), (z, y)\} \rangle$ but is not in $\mathcal{PAIR}$. It is easy to verify that every other previously introduced system is dicut-closed. For these systems we obtain:

Proposition 3.3 *Consider any extraction algorithm for the system $\mathcal{X}$.*

- *If $\mathcal{X} = \mathcal{STAR}$, then the center of the extracted star is a correct process.*
- *If $\mathcal{X} = \mathcal{TREE}$, then the root of the extracted tree is a correct process.*
- *If $\mathcal{X} \in \{\mathcal{SC}, \mathcal{COMPLETE}, \mathcal{RING}, \mathcal{BIC}\}$, then the extraction is exact.*

Proof. For $\mathcal{STAR}$ and $\mathcal{TREE}$, all the dicut reductions of the extracted graph contain at least respectively the center and the root, then the restriction of the extracted graph contains at least these nodes, proving that they are correct processes.

There is no dicut for a strongly connected graph. Hence in $\mathcal{SC}$, there is no dicut reduction then by the Closure property the subgraph induced by the set of correct processes of the extracted graph is the extracted graph itself. $\mathcal{COMPLETE}$, $\mathcal{RING}$, and $\mathcal{BIC}$ are particular cases of systems only composed of strongly connected timeliness graphs. □

An immediate consequence of Proposition 3.3 is that any extraction algorithm gives an implementation of eventual leader election (failure detector Ω) for systems $\mathcal{STAR}$ and $\mathcal{TREE}$ as well as an implementation of failure detector $\Diamond\mathcal{P}$ for systems $\mathcal{COMPLETE}$, $\mathcal{RING}$, $\mathcal{SC}$ and $\mathcal{BIC}$.

Due to the lack of space, the proofs of the two following propositions have been moved in the appendix. In the first proposition we show that extraction is not always possible. Actually, in the proof we exhibit some non dicut-closed systems, namely $\mathcal{PAIR}$, where no extraction algorithm can be implemented.

Proposition 3.4 *There exist some systems $\mathcal{X}$ for which there is no extraction algorithm.*

In the next section we show that for all dicut-closed systems there is an extraction algorithm. For systems like $\mathcal{STAR}$, $\mathcal{TREE}$ and $\mathcal{PAIR}$, there exists no *exact* extraction algorithm.

Proposition 3.5 *There exist some systems $\mathcal{X}$ for which there is an extraction algorithm and there is no exact extraction algorithm.*

4 An Extraction Algorithm

The aim of this section is to show that the dicut-closed property of a system is sufficient to solve the extraction problem. To that end, we propose in Figure 1 an extraction algorithm, called $\mathcal{A}(\mathcal{X})$, for dicut-closed systems $\mathcal{X}$.

The basic idea of Algorithm $\mathcal{A}(\mathcal{X})$ is to make processes select a graph that is compatible with the timeliness graph of the run. For this, each process maintains for each graph x in $\mathcal{X}$ an *accusation counter* $Acc[x]$. This counter infinitely grows if some correct process is not in x or if some directed edge of x is not timely. Then, $Acc[x]$ is bounded if and only if x contains all correct processes and all timely links between pairs of correct processes.

We implement accusation counters as follows. A process regularly blames all the graphs in $\mathcal{X}$ in which it is not a node: it increments the accusation counters of all these graphs. Note that if the process is correct this accusation is justified and if the process is not correct, after some time, the process being dead stops to increment the accusation counters. Moreover, each process regularly sends on its outgoing links $alive$ messages. Each process maintains an estimate of the communication delays for each incoming link ($\Delta[q]$ for the incoming link (q, p)). If it does not receive $alive$ messages within these estimates on some incoming link it blames all timeliness graphs in $\mathcal{X}$ containing this link (*i.e.*, increments the accusation counters for these graphs). As the estimate of the communication delay may be too short, each time it is exceeded the process increases it for the link. In this way, if the link is timely, at some time the estimate will be greater than the bound on communication delay.

The accusation counters are broadcast by reliable broadcasts. Each time a process receives a new value of accusation counter it updates its own accusation counter to the maximum of the received values and its current values. Hence, if some timely graph stops to be blamed then all correct processes eventually agree on the value of its accusation counter.

By selecting the graph G with the lowest accusation value (to break ties, we assume a total order among the graphs of $\mathcal{X}$) if any, correct processes eventually agree on the same timeliness graph of $\mathcal{X}$, moreover we can prove that this graph contains (1) all the correct processes, and (2) all edges between correct processes are timely links. As a consequence, the Convergence, the Compatibility and the Validity properties of the extraction algorithm are ensured. Nevertheless, this graph can also contain faulty processes and edges between correct and faulty processes.

Consider now the Closure property. If G contains only correct processes then the Closure property is trivially satisfied. Otherwise, G contains $Correct(r)$ and a set F of faulty processes. In this case, $(Correct(r), F)$ is a dicut reduction of G: Indeed if there is an edge in G from a faulty process q to a correct process p, eventually the process p stops to receive messages from q and the accusation counter of G grows infinitively often. Hence, in all cases, the Closure property is satisfied.

Hence, if $\mathcal{X}$ is dicut-closed, Algorithm $\mathcal{A}(\mathcal{X})$ extracts a graph in $\mathcal{X}$. Moreover from Proposition 3.3, if all the graphs of $\mathcal{X}$ are strongly connected then the algorithm exactly extracts a graph in $\mathcal{X}$.

In the algorithm, each process p uses local timers, one per process. The timer of p dedicated to q is set (by setting `settimer`(q) to a positive value) to a time interval rather than absolute time. The timer is decremented until it expires. When the timer expires `timerexpire`(q) becomes $true$. Note that a timer can be restarted before it expires.

In the algorithm, we denote by $\prec$ the total order relation on $\mathcal{X}$ and by $\prec_{lex}$ (see Line 2) the total order relation defined as follows: $\forall x, y \in \mathcal{X}, \forall c_x, c_y \in \mathbb{N}, (c_x, x) \prec_{lex} (c_y, y) \equiv [c_x < c_y \vee (c_x = c_y \wedge x \prec y)]$.

```
Code for each process p
 1: Procedure updateExtractedGraph()
 2:     G ← x such that (Acc[x], x) = min_{≺lex} {(Acc[x'], x') such that x' ∈ X}

 3: On initialization:
 4: for all x ∈ X do Acc[x] ← 0
 5: for all q ∈ Π \ {p} do
 6:     Δ[q] ← 1
 7:     settimer(q) ← Δ[q]
 8: updateExtractedGraph()
 9: start tasks 1 and 2

10: task 1:
11:     loop forever
12:         send⟨alive⟩ to every q ∈ Π \ {p} every K time
13:         rbroadcast⟨ACC,⊥,p⟩ every K time      /* to accuse graphs that do not contain p */
14: task 2:
15:     upon receive⟨alive⟩ from q do
16:         settimer(q) ← Δ[q]
17:     upon timerexpire(q) do
18:         rbroadcast⟨ACC, q, p⟩      /* to accuse graphs that contain the link (q, p) */
19:             Δ[q] ← Δ[q] + 1
20:             settimer(q) ← Δ[q]
21:     upon rdeliver⟨ACC,q,h⟩ do      /* information from h */
22:         for all x ∈ X do
23:             if q = ⊥ then
24:                 if h ∉ Node(x) then Acc[x] ← Acc[x] + 1
25:             else
26:                 if (q, h) ∈ Edge(x) then Acc[x] ← Acc[x] + 1
27:         updateExtractedGraph()
```

Figure 1: Algorithm $\mathcal{A}(\mathcal{X})$ extracts a graph in $\mathcal{X}$

A sketch of the correctness proof of $\mathcal{A}(\mathcal{X})$ is given below. In this sketch, we consider a run r of $\mathcal{A}(\mathcal{X})$ in dicut-closed system $\mathcal{X}$. We will denote by var_p^t the value of var of process p at time t.

We first notice that all variables $Acc_p[x]$ are monotonically increasing:

Lemma 4.1 *For all times t and t' such that $t \geq t'$, for all processes p, for all graphs x in $\mathcal{X}$, $Acc_p^t[x] \geq Acc_p^{t'}[x]$.*

Let $\sup(Acc_p[x])$ be the supremum of $Acc_p^t[x]$ for all t, we say that $Acc_p[x]$ is unbounded if $\sup(Acc_p[x])$ is equal to ∞ and bounded otherwise. As $Acc_p[x]$ is also updated by reliable broadcast each time some process q modifies $Acc_q[x]$ we have:

Lemma 4.2 *For all correct processes p and q, for all graphs x in $\mathcal{X}$, $\sup(Acc_p[x]) = \sup(Acc_q[x])$*

Let $\sup(Acc[x])$ be the supremum $\sup(Acc_p[x])$ over all correct process p of $Acc_p[x]$, then $\sup(Acc[x])$ is well-defined. If there is a least one $x \in \mathcal{X}$ such that $\sup(Acc[x])$ is bounded, then $\min\{\sup(Acc[x])|x' \in \mathcal{X}\}$ is finite, hence G the graph such that $(Acc[G], G) = \min_{\prec_{lex}}\{(Acc[x'], x')|x' \in \mathcal{X}\}$ is well defined. Then all correct processes converge to the same graph:

Lemma 4.3 *If there exists x in $\mathcal{X}$ such that $\sup(Acc[x])$ is bounded then there is a time after which for every correct process p, G_p is G.*

Now prove the Compatibility property. Consider any timeliness graph compatible with $T(r)$, and assume that $x \in \mathcal{X}$, then there is a time t_0 after which all faulty processes are dead and the estimates of communication delays are greater than the bounds of communication delays of timely links of the run. After time t_0, (1) as x contains all correct processes, no process will blame x because it is not a node of x, and (2) as all edges of x are timely, no process will blame x for one of its edges then:

Lemma 4.4 *If x in $\mathcal{X}$ is compatible with $T(r)$, then $\sup(Acc[x])$ is bounded.*

Reciprocally, let x be a timeliness graph of $\mathcal{X}$ that is not compatible with the run. If process p is not correct there is a time t after which it does not send any $alive$ message, and there is a time after the timers on p expire forever for all correct processes, then if p is a node of some $x \in \mathcal{X}$, $Acc_p[x]$ is incremented infinitely often and $\sup(Acc[x]) = \infty$. In the same way if (p, q) is not timely, by the fifo property of the link, the timer for p expires infinitely often for process q and if (p, q) is an edge of x then $Acc_q[x]$ is incremented infinitely often and $\sup(Acc[x]) = \infty$.

Then:

Lemma 4.5 *For every x in $\mathcal{X}$, if $\sup(Acc[x])$ is bounded then $x[Correct(r)]$ is compatible with $T(r)$.*

Hence:

Lemma 4.6 *(Compatibility) $G[Correct(r)]$ is compatible with $T(r)$.*

It remains to prove that G satisfies the Closure property: $G[Correct(r)]$ is a dicut reduction of G or is equal to G. As $G[Correct(r)]$ is compatible with $T(r)$, we have:

Lemma 4.7 $Correct(r) \subseteq Node(G)$.

Let $F = Node(G) - Correct(r)$. If F is empty the Closure property is trivially ensured. Consider now the case where F is not empty. F contains only faulty processes and $(Correct(r), F)$ is a partition of $G(Node)$. If there is an edge in $Edge(G)$ from a faulty process q to a correct process p, eventually the process p never receives a message from q and the accusation counter of G will be unbounded, contradicting the choice of G. So, we have:

Lemma 4.8 *If $F \neq \emptyset$ then $Edge(G) \cap (F \times Correct(r)) = \emptyset$.*

Hence, $(Correct(r), F)$ is a dicut of G.

Lemma 4.3 and Lemma 4.4 prove the Convergence property, Lemma 4.6 proves the Compatibility property and Lemma 4.8 proves the Closure property. Moreover, G is clearly in $\mathcal{X}$ proving the Validity. Proposition 3.3 shows that the extraction is exact when all graphs of $\mathcal{X}$ are strongly connected. Hence, we can conclude with the following theorem:

Theorem 4.9 *Let $\mathcal{X}$ be a dicut-closed system. Algorithm $\mathcal{A}(\mathcal{X})$ extracts a graph in $\mathcal{X}$. Moreover if all graphs of $\mathcal{X}$ are strongly connected, Algorithm $\mathcal{A}(\mathcal{X})$ exactly extracts a graph in $\mathcal{X}$.*

5 An Efficient Extraction Algorithm

In this section, we propose another extraction algorithm called $\mathcal{AF}(\mathcal{X})$ (Figures 2 and 3). This algorithm is efficient meaning that the (correct) processes eventually only send messages along the edges of the extracted graph.

$\mathcal{AF}(\mathcal{X})$ (exactly) extracts a timeliness graph from system $\mathcal{X}$, where (1) $\mathcal{X}$ is dicut-closed and (2) for all graphs $g \in \mathcal{X}$ there is some process p, called *root*, such that there is a directed path from p to every node of g. For example, $\mathcal{TREE}$ and $\mathcal{RING}$ systems have this property.

In the following, we refer to these systems as *dicut-closed systems with a root*. For every graph g in $\mathcal{X}$, the function $root(g)$ returns a root of g.

In the algorithm, every process p stores several values concerning the graphs $x \in \mathcal{X}$ such that $root(x) = p$: (1) $Acc[x]$ is the accusation counter of x whose goal is the same as in Algorithm 1, (2) $Prop[x]$ is a *proposition counter* whose goal will be explained later, and (3) $\Delta[x]$ gives the expected time for a message to go from p (the root of the x) to all the nodes of x.

Every process also maintains a set variable $Candidates$. Each element of this set is a 4-tuple composed of a graph x of $\mathcal{X}$ and the newest values of $Acc[x]$, $Prop[x]$, and $\Delta[x]$ known by the process (the exact values are maintained at $root(x)$). Each element in this set is called *candidate* and each process selects its extracted graph among the graphs in the candidate elements.

As in Algorithm 1:

(1) Each process p sends $alive$ messages on its outgoing links and monitors its incoming links. However, we restrain here the $alive$ message sendings: process p sends $alive$ messages on its outgoing link (p, q) only if (p, q) is in a graph candidate.

(2) A graph candidate is blamed if (a) a correct process is not in the graph or (b) a process receives an out of date message through one of its incoming links. In both cases the candidate is definitively removed from the $Candidates$ sets of all processes. To achieve this goal the process sends an accusation message (ACC) using a reliable broadcast and uses an array $Heard$ that ensures that an identical candidate (that is, the same graph with the same accusation and proposition values) can never be added again. Moreover, upon delivery of an accusation message for graph x, $root[x]$ increments $Acc[x]$.

We now present different mechanisms used to obtain the efficiency.

For all graphs $x \in \mathcal{X}$, only the process $root(x)$ is allowed to propose x as a candidate to the rest. Each process p stores its better candidate in its variable me, that is, the least blamed graph x such that $root(x) = p$.

- If a process finds in $Candidates$ a better candidate than me, it removes me from $Candidates$.

- If a process finds that me is better, it adds me to $Candidates$ and sends a new message containing me (1) to all processes that are not in $Node(me)$, and (2) to immediate successors of p in me. The immediate successors in me add me to their $Candidates$ set and relay the new message, and so on. By the reliability of the links, every correct process that is not in me eventually receives this message and blames me.

These mechanisms are achieved by the procedure $updateExtractedGraph()$. This procedure is called each time a graph candidate is blamed or a new candidate is proposed. Note that the $Candidates$ set is maintained with the set $OtherCand$ (the candidates of other processes), a boolean $Local$ that is true when the process has a candidate, and me, the graph candidate.

A process p may give up a candidate without this candidate being blamed: in this case, p is the root of the candidate, it finds a better candidate in $OtherCand$, and removes me from $Candidates$. Then, p must not increment $Acc[me]$ when it receives accusations caused by this removing, indeed these accusations are not due to delayed messages. That is the goal of the proposition counter ($Prop$): in $Prop[x]$, $root(x)$ counts the number of times it proposes x as candidate and includes this value in each of its new messages (to inform other process of the current value of the counter). Hence, when q wants to blame x, it now includes its own view of $Prop[x]$ in the accusation message. This accusation will be considered as legitimate by $root[x]$ (that is, will cause an increment of $Acc[x]$) only when the proposition counter inside the message matches $Prop[x]$. Also, whenever $root[x]$ removes x from $Candidates$, $root[x]$ increments $Prop[x]$ and does not send the new value to the other processes. In this way accusations due to this removing will be ignored.

For any timely candidate, the accusation counter will be bounded and its proposition counter increased each time it is proposed. In this way the graph with the smallest accusation and proposition values eventually remains forever in the $Candidates$ set of all correct processes and it is chosen as extracted graph. (This is done in the procedure $updateExtractedGraph()$.) Moreover, eventually all other candidates are given up and it remains only this graph in $Candidates$. In this way, only $alive$ messages are sent and they are sent along the directed edges of the extracted graph ensuring the efficiency.

```
Code for each process p
 1: Procedure updateExtractedGraph()
 2:     Let (a_min, min) = min_{≺lex}{(acc, c) such that (c, acc, −, −) ∈ OtherCand} ∪ {(∞, ∞)}
 3:     if (a_min, min) < (Acc[me], me) ∧ Local then        /* Give up me */
 4:         rbroadcast⟨ACC,me,Acc[me],Prop[me],Δ[me]⟩
 5:         Prop[me] ← Prop[me] + 1
 6:         Local ← false
 7:     Candidates ← OtherCand
 8:     me ← x such that (a, x) = min_{≺lex}{(acc, c) such that c ∈ X ∧ root(c) = p}
 9:     if (Acc[me], me) < (a_min, min) ∧ Local = false then        /* Propose me */
10:         Local ← true
11:         Candidates ← Candidates ∪ {(me, Acc[me], Prop[me], Δ[me])}
12:         send⟨new,me,Acc[me],Prop[me],Δ[me]⟩ to every process not in Node(me)
13:         for all h ∈ Π \ {p} do
14:             if (h,p)∈ Edge(me) then
15:                 Δ[h]← max(Δ[h], Δ[me])
16:                 settimer(h) ← Δ[h]
17:             if (p,h)∈ Edge(me) and h ≠ root(me) then
18:                 send⟨new,me, Acc[me], Prop[me], Δ[me]⟩ to h
19:     G ← x such that (a, x) min_{≺lex}{(a′, x′) such that (x′, a′, p′, d′) ∈ Candidates}
```

Figure 2: Procedure updateExtractedGraph of Algorithm $\mathcal{AF}(\mathcal{X})$

A sketch of the correctness proof of $\mathcal{AF}(\mathcal{X})$ is given in the appendix. Then, we can conclude with the following theorem:

Theorem 5.1 *Let $\mathcal{X}$ be a dicut-closed system with a root. Algorithm $\mathcal{A}(\mathcal{X})$ efficiently extracts a graph in $\mathcal{X}$. Moreover if all graphs of $\mathcal{X}$ are strongly connected, Algorithm $\mathcal{A}(\mathcal{X})$ efficiently and exactly extracts a*

```
Code for each process p
20: On initialization:
21: for all x ∈ 𝒳 such that root(x) = p do
22:       Acc[x] ← 0; Prop[x] ← 0; Δ[x] ← n
23: for all x ∈ 𝒳 such that root(x) ≠ p do Heard[x] ← (−1, −1)
24: for all q ∈ Π \ {p} do Δ[q] ← 1
25: OtherCand ← ∅
26: Local ← false
27: me ← min{x such that x ∈ 𝒳 ∧ root(x) = p}
28: updateExtractedGraph()
29: start tasks 1 and 2

30: task 1:
31:     loop forever
32:         send⟨alive⟩ to every process q such that ∃(x,-,-,-)∈ Candidates and (p, q) ∈ Edge(x) every K time

33: task 2:
34:     upon receive⟨alive⟩ from q do
35:         settimer(q) ← Δ[q]

36:     upon timerexpire(q) do        /* Link (q, p) is not timely, blame all candidates that contains (q, p) */
37:         for all (x, a, pr, d) ∈ OtherCand such that (q, p) ∈ Edge(x) do
38:             rbroadcast⟨ACC,x,a,pr,d⟩
39:         if (q, p) ∈ Edge(me) then
40:             rbroadcast⟨ACC,me,Acc[me],Prop[me],Δ[me]⟩

41:     upon receive⟨new, x, a, pr, d⟩ from q do        /* Proposition of a new candidate */
42:         if p ∉ Node(x) then        /* Blame x that does not contain p */
43:             rbroadcast⟨ACC,x,a,pr⟩
44:         else
45:             newCand ← false
46:             if (x, −, −, −) ∉ OtherCand and Heard(x) < (a, pr) then        /* New candidate */
47:                 newCand ← true
48:             if ∃(x, a_c, pr_c, d_c) ∈ OtherCand with (a_c, pr_c) < (a, pr) then        /* New candidate */
49:                 OtherCand ← OtherCand \ (c, a_c, pr_c, d_c)
50:                 newCand ← true
51:             if newCand then
52:                 OtherCand ← OtherCand ∪ (x, a, pr, d)
53:                 updateExtractedGraph()
54:                 Heard[x] ← (a, pr)
55:                 for all h ∈ Π \ {p} do
56:                     if (h,p)∈ Edge(x) then
57:                         Δ[h]← max(Δ[h], d)
58:                         settimer(h)← Δ[h]
59:                     if (p,h)∈ Edge(x) and h ≠ root(x) then send⟨new, x, a, pr, d⟩ to h

60:     upon rdeliver⟨ACC,x,a,pr,d⟩ do
61:         if root(x) = p then
62:             if x = me ∧ a = Acc[me] ∧ pr = Prop[me] then        /* Check if the accusation is up to date */
63:                 Acc[me] ← Acc[me] + 1; Δ[me] ← Δ[me] + 1
64:                 Local ← false
65:         else
66:             OtherCand ← OtherCand \ (x, a, pr, d)
67:             if Heard[x] < (a, pr) then Heard[x] ← (a, pr)
68:         updateExtractedGraph()
```

Figure 3: Algorithm $\mathcal{AF}(\mathcal{X})$ that efficiently extracts a graph in $\mathcal{X}$

graph in $\mathcal{X}$.

6 Conclusion

Failure detector implementations in partially synchronous models generally use the timeliness properties of the system to approximate the set of correct (or faulty) processes. In some way, the extraction problem is a kind of generalization: instead of only searching the set of correct processes, here we try to extract also

information about the timeliness of links. Besides, our solutions are based on already existing mechanisms used in failure detectors implementations as in [2, 3].

Information about the timeliness of links is useful for efficienecy of fault-tolerant algorithms. In particular, in any extracted graph, any path between a pair of correct processes is only constituted of timely links. This property is particulary interesting to get efficient routing algorithms.

We gave an extraction algorithm for dicut-closed set of timeliness graphs. Moreover, we proved that the extraction is exact when all the timeliness graphs are also strongly connected.

Given dicut-closed timeliness graphs that contains a root, we shown how to efficiently extract a graph from it. By efficiency we mean giving a solution where eventually messages are only sent over the links of the extracted graph.

It is important to note that the main purpose of the algorithms we proposed is to show the feasability of the extraction under some conditions. So, the complexity of our algorithms was not the main focus of this paper.

As a consequence, our algorithms are somehow unrealistic because of their high complexity. Giving more practical solutions will be the purpose of our future works.

Acknowledgments

We are grateful to members of the *GRAPH* team of the *LIAFA* Lab for the helpful discussions and their interesting suggestions.

References

[1] Marcos K. Aguilera, Sam Toueg, and Boris Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *DISC '99: Proceedings of the thirteenth International Symposium on Distributed Computing*, pages 13–33, LNCS vol. 1693. Springer-Verlag, September 1999.

[2] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC*, pages 306–314, 2003.

[3] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In Soma Chaudhuri and Shay Kutten, editors, *PODC*, pages 328–337. ACM, 2004.

[4] Rida A. Bazzi and Gil Neiger. Simulating crash failures with many faulty processors (extended abstract). In *6th International Workshop on Distributed Algorithms (WDAG '92)*, volume 647 of *Lecture Notes in Computer Science*, pages 166–184. Springer, 1992.

[5] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[6] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[7] Benny Chor and Brian A. Coan. A simple and efficient randomized byzantine agreement algorithm. *IEEE Trans. Software Eng.*, 11(6):531–539, 1985.

[8] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. A realistic look at failure detectors. In *DSN*, pages 345–353. IEEE Computer Society, 2002.

[9] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 338–346, 2004.

[10] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, April 2005.

[11] Danny Dolev, Cynthia Dwork, and Larry J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[12] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[13] Jonathan Eisler, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector to solve nonuniform consensus. *Distributed Computing*, 19(4):335–359, 2007.

[14] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[15] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[16] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In *DISC '06: Proceedings of the twentieth International Symposium on Distributed Computing*, pages 399–412, LNCS vol. 4167. Springer-Verlag, September 2006.

[17] Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001.

[18] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, 1994.

[19] Joseph Y. Halpern and Aleta Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 73–82, 1999.

[20] Nancy A. Lynch, Yishay Mansour, and Alan Fekete. Data link layer: Two impossibility results. In *Symposium on Principles of Distributed Computing*, pages 149–170, 1988.

[21] Michel Raynal and Corentin Travers. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In Alexander A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2006.

[22] Piotr Zielinski. Anti-omega: the weakest failure detector for set agreement. Technical Report UCAM-CL-TR-694, Computer Laboratory, University of Cambridge, Cambridge, UK, July 2007.

A Appendix

A.1 Proof of Proposition 3.4

Proposition 3.4 *There exists some systems $\mathcal{X}$ for which there is no extraction algorithm.*

Sketch of Proof.

Assume there is an extraction algorithm $\mathcal{A}$ for $\mathcal{PAIR}$ with 5 processes.

Consider a run r of $\mathcal{A}$ in system $\mathcal{PAIR}$ with $T(r) = \langle\{p_1, p_2, p_3, p_4, p_5\}, \{(p_1, p_2), (p_2, p_1), (p_3, p_4), (p_4, p_3)\}\rangle$. To satisfy the properties of the extraction, $\langle\{p_1, p_2, p_3, p_4, p_5\}, \{(p_1, p_2), (p_2, p_1)\}\rangle$ or $\langle\{p_1, p_2, p_3, p_4, p_5\}, \{(p_3, p_4), (p_4, p_3)\}\rangle$ must be extracted from the run r. There is a time t_1 after which r converges for example to $\langle\{p_1, p_2, p_3, p_4, p_5\}, \{(p_1, p_2), (p_2, p_1)\}\rangle$.

Consider now run r' of $\mathcal{A}$ in system $\mathcal{PAIR}$ with $T(r') = \langle\{p_3, p_4, p_5\}, \{(p_3, p_4), (p_4, p_3)\}\rangle$ such that r and r' are indistinguishable until time t_1 and p_1 and p_2 crash in r' at time $t_1 + 1$. There is a time t_2 after which r' converges to a graph with the directed edges $\{(p_3, p_4), (p_4, p_3)\}$.

Consider now that in r all messages from p_1 and p_2 to $\{p_3, p_4, p_5\}$ sent after time t_1 are delayed after time t_2. For p_5, the runs r and r' are indistinguishable until t_2. So, at time t_2, p_5 outputs a graph with directed edges $\{(p_3, p_4), (p_4, p_3)\}$.

Now consider run r'' of $\mathcal{A}$ in system $\mathcal{PAIR}$ with $T(r'') = \langle\{p_1, p_2, p_5\}, \{(p_1, p_2), (p_2, p_1)\}\rangle$ such that r and r'' are indistinguishable until time t_2 and p_3 and p_4 crash in r'' at time $t_2 + 1$. There is a time t_3 after which r'' converges to a graph with the directed edges $\{(p_1, p_2), (p_2, p_1)\}$.

Consider again that in the run r all messages from p_3 and p_4 to $\{p_1, p_2, p_5\}$ sent after time t_2 are delayed after t_3. For p_5 the runs r and r'' are indistinguishable. So, at time t_3, p_5 outputs a graph with directed edges $\{(p_1, p_2), (p_2, p_1)\}$.

Inductively, we can construct the run r in such a way that p_5 alternates forever between a graph with directed edges $\{(p_1, p_2), (p_2, p_1)\}$ and a graph with directed edges $\{(p_3, p_4), (p_4, p_3)\}$ and never converges definitively. This contradicts the existence of an algorithm that extracts a graph in $\mathcal{PAIR}$. □

A.2 Proof of Proposition 3.5

Proposition 3.5 *There exists some systems $\mathcal{X}$ for which there is an extraction algorithm and there is no exact extraction algorithm.*

Sketch of Proof. Consider the system $\mathcal{TREE}$ with 3 processes. We prove in the next section that there is an extraction algorithm for this system. Assume there is an *exact* extraction algorithm $\mathcal{A}$ for this system.

Consider a run r of $\mathcal{A}$ in this system with $T(r) = \langle\{p_1, p_2, p_3\}, \{(p_1, p_2), (p_1, p_3)\}\rangle$. To satisfy the properties of the exact extraction, there is a time t_1 after which the graph $\langle\{p_1, p_2, p_3\}, \{(p_1, p_2), (p_1, p_3)\}\rangle$ is extracted.

Consider now run r' of $\mathcal{A}$ in system $\mathcal{TREE}$ with $T(r') = \langle\{p_1, p_2\}, \{(p_1, p_2)\}\rangle$ such that r and r' are indistinguishable until time t_1 and p_3 crashes in r' at time $t_1 + 1$. There is a time t_2 after which r' converges to $\langle\{p_1, p_2\}, \{(p_1, p_2)\}\rangle$.

Consider now that in r all messages from p_3 to $\{p_1, p_2\}$ sent after time t_1 are delayed after time t_2. For p_1, the run r and r' are indistinguishable until t_2. So, at time t_2, p_1 outputs $\langle\{p_1, p_2\}, \{(p_1, p_2)\}\rangle$.

Inductively, we can construct the run r in such a way that p_1 alternates forever between a graph $\langle\{p_1, p_2, p_3\}, \{(p_1, p_2), (p_1, p_3)\}\rangle$ and a graph $\langle\{p_1, p_2\}, \{(p_1, p_2)\}\rangle$ and never converges definitively. This contradicts the existence of an algorithm that exactly extracts a graph in $\mathcal{TREE}$. □

A.3 Proof of Theorem 5.1

In this section, we propose a sketch of the correctness proof of the efficient extraction algorithm $\mathcal{AF}(\mathcal{X})$ (Figures 2 and 3). In this sketch, we consider a run r of $\mathcal{AF}(\mathcal{X})$ in dicut-closed system with a root, $\mathcal{X}$. We will denote by var_p^t the value of var_p at time t.

We first notice that all variables $Acc[x]$ and $Prop[x]$ can only be modified by the process $root(x)$ and are increasing:

Lemma A.1 *For all time t and t', $t \geq t'$, for all processes p, for all graphs x in $\mathcal{X}$ such that $p = root(x)$, $Acc_p^t[x] \geq Acc_p^{t'}[x]$ and $Prop_p^t[x] \geq Prop_p^{t'}[x]$.*

Consider a graph x such that its root p crashes. Eventually, every process q such that $x \in OtherCand$ and $(p, q) \in Edge(x)$ reliably broadcasts an accusation for x. This way, x is removed from the $OtherCand$ set of any correct process and never more added (because p is crashed), hence:

Lemma A.2 *If p is faulty, there exists a time t such that for all graphs x of $\mathcal{X}$ with $root(x) = p$, for all correct processes q in r, for all $t' \geq t$: $x \notin OtherCand_q^{t'}$.*

As r is a run of $\mathcal{X}$, there exists some timeliness graph o in $\mathcal{X}$ such that o is compatible with $T[r]$. In this case, $Nodes(o) = Correct(r)$ and the process $root(o)$ is a correct process:

Lemma A.3 *There exists a timeliness graph o of $\mathcal{X}$ such that o is compatible with $T(r)$ and $root(o)$ is a correct process.*

Moreover:

Lemma A.4 *Let o be a timeliness graph of $\mathcal{X}$ such that $o[Correct(r)]$ is a compatible with $T(r)$ and $root(o)$ is a correct process: $Acc_{root(o)}[o]$ is bounded.*

For all correct processes p, for all graphs x in $\mathcal{X}$ with $root(x) = p$, let $A[x]_p$ be the largest value of $Acc[x]_p$ in r (∞ if $Acc[x]_p$ is unbounded). Let g to be the graph with the smallest $A[g]_p$ (break ties by the total order on graphs). Let C be the value of $A[g]_p$.

Note that from Lemma A.3 and Lemma A.4, $C < \infty$. Moreover, by construction of g, $root(g)$ is a correct process, $root(g)$ eventually elects g forever ($me_{root(g)} = g$), and as a consequence $Prop[g]_{root(g)}$ becomes constant:

Lemma A.5 *There exists a time after which $me_{root(g)} = g$.*

Lemma A.6 *There exists a time after which $Prop[g]_{root(g)}$ stops changing.*

Let P be the largest value of the proposition counter of g ($Prop[g]$). The following three lemmas are immediate consequences of Lemma A.5:

Lemma A.7 *For every correct process $p \neq root(g)$, there exists a time after which $g \in OtherCand_p$.*

Lemma A.8 *There exists a time after which $me_{root(g)} = g$ and $Local_{root(g)} = true$ and $OtherCand_{root(g)} = \emptyset$.*

Lemma A.9 *For every correct process $p \neq root(g)$, there exists a time after which $OtherCand_p = \{g\}$ and $Local_p = false$.*

From Lemmas A.8 and A.9, the algorithm converges to a graph of $\mathcal{X}$:

Lemma A.10 *There exists a timeliness graph $x \in \mathcal{X}$ (actually g) such that every correct process q outputs x forever.*

From Lemma A.8 and Lemma A.9, we can deduce that the algorithm is efficient:

Lemma A.11 *There is a time after which every correct process p sends messages only to the process q such that there is a directed edge (p, q) in $Edge(g)$.*

From the Lemma A.10, we deduce the Convergence and the Validity properties.

It remains to prove that g satisfies the properties of the approximation: (1) $g[Correct(r)]$ is compatible with $T[r]$, and (2) $g[Correct(r)]$ is a dicut reduction of g or is equal to g.

When $root(g)$ sets $Local$ to true and me to $(g, C, P, -)$, it sends a message new to all processes (recall that C the final value of the accusation counter of g and P the final value of its the proposition counter.). As the links are reliable, all correct processes eventually receives this message. If a correct process q is not in $Node(g)$, it reliably broadcasts an accusation message ACC. When process $root(g)$ delivers such a broadcast, it increments the accusation counter of g contradicting the fact that $Acc[g]$ is bounded by C, hence:

Lemma A.12 $Correct(r) \subseteq Node(g)$.

When a correct process receives this new message, it sends $\langle alive \rangle$ to every process q such that (p, q) in $Edge(g)$. And it monitors all incoming links (q, p) such that (q, p) in $Edge(g)$. If there is a link (a, b) of $Edge(g)$ between two correct processes a and b, then a sends regularly $alive$ message to b. By construction of g, b never blames g, then b receives no out of date message. By the FIFO property of the link, the link is timely:

Lemma A.13 $g[Correct(r)]$ *is compatible with* $T[r]$.

By Lemma A.12, $Node(g) = Correct(r) \cup F$.

If F is empty the Closure property is trivially ensured. We now consider the case where F is not empty. F contains only faulty processes. If there is an edge in $Edge(g)$ from a faulty process q to a correct process p, eventually the process p stops receiving messages from q and the accusation counter of g will be incremented, which contradicts the fact that the accusation counter of g remains equal to C forever. So we have:

Lemma A.14 *If* $F \neq \emptyset$ *then* $Edge(g) \cap (F \times Correct(r)) = \emptyset$.

We showed the Convergence (Lemma A.10), the Validity (Lemma A.10), the Compatibility (Lemma A.13), the closure (Lemma A.14), and the Efficiency (Lemma A.11). Moreover, Proposition 3.3 shows the exact extraction when all graphs of $\mathcal{X}$ are strongly connected. Hence, we can conclude with the following theorem:

Theorem 5.1 *Let* $\mathcal{X}$ *be a dicut-closed system with a root. Algorithm* $\mathcal{A}(\mathcal{X})$ *efficiently extracts a graph in* $\mathcal{X}$*. Moreover if all graphs of* $\mathcal{X}$ *are strongly connected, Algorithm* $\mathcal{A}(\mathcal{X})$ *efficiently and exactly extracts a graph in* $\mathcal{X}$*.*