Technical Report

Identification: DCC-UFBA 12-2006

Title:
Knowledge Connectivity vs. Synchrony Requirements for Fault-Tolerant Agreement in Unknown Networks

Authors:
Fabíola Gonçalves Pereira Greve
Sébastien Tixeuil

Contact author:
Fabíola Greve
fabiola@dcc.ufba.br

Departamento de Ciência da Computação
Instituto de Matemática
Universidade Federal da Bahia
Av. Adhemar de Barros, S/N, Campus de Ondina
40.170-110 - Salvador - BA - BRASIL

Phone: +55 (71) 3263 6309
Fax: +55 (71) 3263 6276
Knowledge Connectivity vs. Synchrony Requirements for Fault-Tolerant Agreement in Unknown Networks

Fabíola Greve*
fabiola@dcc.ufba.br
* DCC - Computer Science Department
Federal University of Bahia
40170-110 Bahia, Brasil

Sébastien Tixeuil†
tixeuil@lri.fr
† LRI - Laboratoire de Recherche en Informatique
Université Paris-Sud
91405 Orsay Cedex, France

Abstract

In self-organizing systems, such as mobile ad-hoc and peer-to-peer networks, consensus is a fundamental building block to solve agreement problems. It contributes to coordinate actions of nodes distributed in an ad-hoc manner in order to take consistent decisions. It is well known that in classical environments, in which entities behave asynchronously and where identities are known, consensus cannot be solved in the presence of even one process crash. It appears that self-organizing systems are even less favorable because the set and identity of participants are not known. We define necessary and sufficient conditions under which fault-tolerant consensus become solvable in these environments. Those conditions are related to the synchrony requirements of the environment, as well as the connectivity of the knowledge graph constructed by the nodes in order to communicate with their peers.

Keywords: fault-tolerant agreement, distributed consensus, self-organizing systems, unknown networks, wireless ad-hoc networks

1 Introduction

Wireless sensor and ad hoc networks (and, in a different context, unstructured peer to peer networks) enable participating entities access to services and informations independently of their location or mobility. This is done by eliminating the necessity of any statically designed infrastructure or any centralized administrative authority. It is in the nature of such systems to be self-organizing, since additionally, entities are allowed to join or leave the network in an arbitrary manner, making the whole system highly dynamic.

Contrarily to traditional (i.e. wired) networks, where processes are aware of network topology and have a complete knowledge of every other participant, in a self-organizing environment with no central authority, the number and processes are not known initially. Yet, even in a classical environment, when entities behave asynchronously, consensus cannot be solved if one of the participants is allowed to crash [6]. Thus, solving consensus when the set of participants is unknown is even more difficult. Nonetheless, due to the essential role of this problem, we study in this paper the conditions that permit to solve consensus in unknown asynchronous networks in spite of participant crashes.

In order to capture the unawareness of self-organizing systems regarding the topology of the network as well as the set of participants, Cachin et al. [1] defined a new problem named CUP (consensus with unknown participants). This new problem keeps the same definition of the classical consensus, except for the expected knowledge about the set of processes in the system. More precisely, they assume that processes are not aware of Π, the set of processes in the system. To solve any non trivial application, processes must somehow get a partial knowledge about the other processes if some cooperation is expected. The participant detector abstraction was proposed to handle this subset of known
processes [1]. They can be seen as distributed oracles that provides hints about the participating processes in the computation. For example, a way to implement participant detectors for mobile nodes is to make use of local broadcasting in order to construct a local view formed by 1-hop neighbors. Based on the initial knowledge graph formed by the participant detectors in the system, Cachin et al. define necessary and sufficient connectivity conditions of this knowledge graph in order to solve CUP in an asynchronous environment but in a fault-free scenario.

In turn, failure detector oracles are an elegant abstraction which encapsulates the extra synchrony necessary to circumvent the impossibility result of fault-tolerant consensus in traditional networks [3, 8]. A failure detector of the class \( \diamond S \) can be seen as an oracle that provides hints on crashed processes [3]. Another failure detector, known as \( \Omega \), is a leader oracle that, eventually, provides processes with the same correct process identity (that is, the same leader) [8]. Both, \( \diamond S \) and \( \Omega \) have the same computational power [5], and they have been proved to be the weakest classes of failure detectors allowing to solve consensus in asynchronous known networks [4]. Those failure detectors may make an arbitrary number of mistakes, but, in spite of their inaccuracy, they will never compromise the safety properties of the consensus protocol that uses them. These consensus protocols are considered indulgent towards these oracles, meaning that they are conceived to tolerate their unreliability during arbitrary periods of asynchrony and instability of the environment. Moreover, any of those indulgent protocols will solve the uniform version of the consensus. The uniform consensus ensures the uniformity of the decision, processes be correct of faulty [7].

In the context of unknown networks, the problem of FT-CUP (fault-tolerant CUP) has been subsequently studied by Cachin et al. [2]. By considering the minimal connectivity requirements over the initial knowledge graph for solving CUP, they identify a perfect failure detector (\( P \)) to fulfill the necessary synchrony requirements for solving FT-CUP. A perfect failure detector never make mistakes and can only be implemented in a synchronous system. Thus, solving FT-CUP in a scenario with the weakest knowledge connectivity demands the strongest synchrony conditions. However, strong synchrony competes with the high dynamism, full decentralization and self-organizing nature of wireless sensor and ad-hoc networks. Moreover, even with a perfect failure detector, when the minimal knowledge connectivity is being considered, the uniform version of FT-CUP cannot be solved in unknown networks [2].

In this paper, we show that there is a trade-off between knowledge connectivity and synchrony for consensus in fault-prone unknown networks. In particular, we focus on solving FT-CUP with minimal synchrony assumption (i.e. the \( \Omega \) failure detector), and investigate necessary and sufficient requirement about knowledge connectivity. If the system satisfies our knowledge connectivity conditions, any of the indulgent consensus algorithms initially designed for traditional networks can be reused to solve FT-CUP as well as uniform FT-CUP.

The remaining of the paper is organized as follows: Section 2 provides the model, notations, and statement of the problem we consider; Section 3 describes necessary and sufficient conditions to solve FT-CUP and uniform FT-CUP with minimal synchrony assumptions. Section 4 provides some concluding remarks.

2 Preliminaries

2.1 Model

We consider a distributed system that consists of a finite set \( \Pi \) of \( n > 1 \) processes, namely, \( \Pi = \{p_1, \ldots, p_n\} \). In a known network, \( \Pi \) is known to every participating process, while in an unknown network, a process \( p_i \) may only be aware of a subset \( \Pi_i \) of \( \Pi \).

Processes communicate by sending and receiving messages through reliable channels (i.e. there is no message creation, corruption, duplication, or loss). A process \( p_i \) may only send a message to another process \( p_j \) if \( p_j \in \Pi_i \). Of course, if a process \( p_i \) sends a message to a process \( p_j \) such that \( p_j \notin \Pi_i \), upon receipt of the message, \( p_j \) may add \( p_i \) to \( \Pi_j \) and send a message back to \( p_i \). We assume the existence of a reliable underlying routing layer, in such a way that if \( p_j \in \Pi_i \), then \( p_i \) can send a message reliably to \( p_j \). There are no assumptions on the relative speed of processes or on message transfer delays, i.e. the system is asynchronous.

A process may fail by crashing, i.e., by prematurely or by deliberately halting (switched off); a crashed process does not recover. A process behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition, a correct process is a process that does not crash. A faulty process is a process that is not correct. Let \( f \) denote the maximum number of processes that may crash in the system. We assume that \( f \) is known to every process.

2.2 Graph Notations

In this paper, we consider directed graphs \( G_{di} = (V, E) \), defined by a set of vertices \( V \) and a set \( E \) of edges \( \{v_1, v_2\} \), which are ordered pairs of vertices of \( V \). A directed path \( v_0 \sim v_k \) is an ordered list of vertices \( v_0, v_1, \ldots, v_k \in V \) such that, for any \( i \in \{0, \ldots, k - 1\} \), \( (v_i, v_{i+1}) \) is an edge of \( E \). The length of this path is \( k \). The distance between two vertices \( u, v \) (denoted by \( d(u, v) \)) is the minimum of the lengths of all directed paths from \( u \) to \( v \) (assuming there exists at least one such path). The out-degree of a vertex \( v \) of \( G_{di} \) is equal to the number of vertices \( u \) such that the edge
for any pair of nodes \((v_i, v_j)\) is in \(E\). A sink is a node with out-degree 0. Throughout the paper, the terms “node”, “vertex” and “process” will be used indistinctly.

A directed graph \(G_{di}(V, E)\) is \(k\)-strongly connected if for any pair of nodes \((v_i, v_j)\), \(v_i\) can reach \(v_j\) through \(k\) distinct node-disjoint paths. In particular, when \(k = 1\), \(G_{di}\) is strongly connected. By Menger’s Theorem [10], it is known that the minimum number of nodes whose removal from \(G_{di}(V, E)\) disconnects nodes \(v_i\) from \(v_j\) is equal to the maximal number of node-disjoint paths from \(v_i\) to \(v_j\). This result leads to the following two observations:

1. For any \(n\) and \(k\), there exists a \(n\)-sized \(k\)-strongly connected directed graph \(G_{di}(V, E)\) such that the removal of \(k\) nodes disconnects the graph.
2. If the directed graph \(G_{di}\) is \(k\)-strongly connected, removing \((k - 1)\) nodes leaves at least one path between any pair of nodes \((v_i, v_j)\). Thus, the graph remains strongly connected.

### 2.3 Synchrony and Knowledge Connectivity for Consensus in Fault-Prone Systems

**Classical Consensus.** The consensus problem is the most fundamental agreement problem in distributed computing. Every process \(p_i\) proposes a value \(v_i\) and all correct processes decide on some unique value \(v\), in relation to the set of proposed values. More precisely, the consensus problem is defined by the following properties [3, 6]:

- **Termination:** every correct process eventually decides some value;
- **Validity:** if a process decides \(v\), then \(v\) was proposed by some process;
- **Agreement:** No two correct processes decide differently.

**Uniform Consensus.** The uniform version of the consensus refines the agreement property so that it is satisfied by every process in the system (be it correct or not). So, it is changed for:

- **Uniform Agreement:** No two processes (correct or not) decide differently.

#### 2.3.1 Failure Detector: a Synchrony Abstraction

A fundamental result in the consensus literature [6] states that even if \(\Pi\) is known to all processes in the system and the number of faulty processes is bounded by 1, consensus can not be solved by a deterministic algorithm in an asynchronous system. To enable solutions, some level of synchrony must be assumed. A nice abstraction to model network synchrony is the failure detector [3]. A failure detector (denoted by FD) can be seen as an oracle that provides hints on crashed processes. Failure detectors can be classified according to the properties (completeness and accuracy) they satisfy. The completeness property refers to the actual detection of crashes; the accuracy property restricts the mistakes a failure detector is allowed to make. The accuracy properties can be ensured only in systems that satisfy some synchrony assumptions. The strong completeness property states that eventually, every process that crashes is permanently suspected by every correct process.

Another approach for encapsulating eventual synchrony consists of extending the system with a leader detector [8], also known as \(\Omega\) [4]. A leader detector is an oracle which eventually provides the same correct process identity to all processes. So, \(\Omega\) ensures that eventually all processes have the same leader. In this paper, we consider two classes of failure detectors:

- **Perfect FD** \((P)\). Those failure detectors never make mistakes. They satisfy the perpetual strong accuracy, stating that no process is suspected before it crashes, and the strong completeness property.
- **Eventually Strong FD** \((\diamond \Omega)\). Those failure detectors can make an arbitrary number of mistakes. Yet, there is a time after which some correct process is never suspected (eventual weak accuracy). Moreover, they satisfy the strong completeness property. It has been proved that \(\diamond \Omega\) and \(\Omega\) have the same computational power [5] and that they are the weakest class of detectors allowing to solve the consensus and the uniform consensus problem in a system of known networks [4]. Relying on \(\diamond \Omega\) and \(\Omega\) failure detectors to solve agreement problems assumes that a majority of processes within the group never fails, i.e., \(f < n/2\).

#### 2.3.2 Participant Detectors: a Knowledge Connectivity Abstraction

With the notable exception of [1, 2], literature on consensus related problems considers that \(\Pi\) is known to every process in the system. In ad hoc and sensor wireless networks, this assumption is clearly unrealistic since processes could be maintained by different administrative authorities, have various wake up times, initializations, failure rates, etc. Of course, some knowledge about other nodes is necessary to run any non trivial distributed algorithm. For example, the use of “Hello” messages (i.e. locally broadcasting your identifier to your vicinity) could be a possible way for each process to get some knowledge about the other processes.

The notion of participant detectors (denoted by PD) has been proposed by [1]. Similarly to failure detectors, they can be seen as distributed oracles that provide information about which processes participate to the system. We denote by \(iPD\) the participant detector of process \(p_i\). When queried by \(p_i\), \(iPD\) returns a subset of processes in \(\Pi\). The information provided by \(iPD\) can evolve between queries. Let \(iPD(t)\) be the query of process \(p_i\) at time \(t\). This query must satisfy the two following properties:
• Information Inclusion. The information returned by the participant detector is non-decreasing over time. \( p_i \in \Pi, t' \geq t : i.PD(t) \leq i.PD(t') \)

• Information Accuracy. The participant detector does not make mistakes. \( \forall p_i \in \Pi, \forall t : i.PD(t) \in \Pi \)

The PD abstraction enriches the system with a knowledge connectivity graph. This graph is directed since knowledge that is given by participation detectors is not necessarily bidirectional (i.e., if \( p_j \in i.PD \), then \( p_i \in j.PD \) does not necessarily hold).

**Definition 1 (Knowledge Connectivity Graph)** Let \( G_{di}(V, E) \) be the directed graph representing the knowledge relation determined by the PD oracle. Then, \( V = \Pi \) and \( (p_i, p_j) \in E \) if and only if \( p_j \in i.PD \), i.e., \( p_i \) knows \( p_j \).

**Definition 2 (Undirected Knowledge Connectivity Graph)** Let \( G(V, E) \) be the undirected graph representing the knowledge relation determined by the PD oracle. Then, \( V = \Pi \) and \( (p_i, p_j) \in E \) if and only if \( p_j \in i.PD \) or \( p_i \in j.PD \).

Based on the induced knowledge connectivity graph, several classes of participant detectors were proposed in [1]:

- **Connectivity PD** (CO). The undirected knowledge connectivity graph \( G \) induced by the PD oracle is connected.
- **Strong Connectivity PD** (SCO). The knowledge connectivity graph \( G_{di} \) induced by the PD oracle is strongly connected.
- **One Sink Reducibility PD** (OSR). The knowledge connectivity graph \( G_{di} \) induced by the PD oracle satisfies the following conditions:
  1. the undirected knowledge connectivity graph \( G \) obtained from \( G_{di} \) is connected;
  2. the directed acyclic graph obtained by reducing \( G_{di} \) to its strongly connected components has exactly one sink.

In this paper, we introduce three new participant detector classes:

- **k-Connectivity PD** (k-CO). The undirected knowledge connectivity graph \( G \) induced by the PD oracle is \( k \)-connected.
- **k-Strong Connectivity PD** (k-SCO). The knowledge connectivity graph \( G_{di} \) induced by the PD oracle is \( k \)-strongly connected.
- **k-One Sink Reducibility PD** (k-OSR). The knowledge connectivity graph \( G_{di} \) induced by the PD oracle satisfies the following conditions:
  1. the undirected knowledge connectivity graph \( G \) obtained from \( G_{di} \) is connected;
  2. the directed acyclic graph obtained by reducing \( G_{di} \) to its \( k \)-strongly connected components has exactly one sink;
  3. consider any two \( k \)-strongly connected components \( G_1 \) and \( G_2 \), if there is a path from \( G_1 \) to \( G_2 \), then there are \( k \) node-disjoint paths from \( G_1 \) to \( G_2 \).

![Figure 1. Knowledge Connectivity Graph Induced for a \( k \)-OSR PD, \( k = 2 \)](image)

Figure 1 illustrates a graph \( G_{di} \) induced by a \( k \)-OSR PD, for \( k = 2 \). Note that there is only one sink component (\( G_3 \)) and that every component \( G_i \) is \( 2 \)-strongly connected.

### 2.4 Problem Statement

In this paper, we concentrate on solving consensus in a fault-prone unknown network. We consider three variants of the problem:

- **CUP (Consensus with Unknown Participants)**. The goal is to solve consensus in an unknown network, where processes may not crash;
- **FT-CUP (Fault-Tolerant CUP)**. The goal is to solve consensus in an unknown network, where up to \( k \) processes (for some constant \( k \)) may crash;
- **Uniform FT-CUP (Uniform Fault-Tolerant CUP)**. The goal is to solve the uniform version of the consensus in an unknown network where up to \( k \) processes may crash.

### 3 Knowledge Connectivity and Synchrony Requirements to Solve FT-CUP

In [1], the CUP problem is investigated in fault free networks, and it is shown that (i) the CO participant detector is necessary to solve CUP; (ii) the SCO participant detector is sufficient to solve CUP, and (iii) the OSR participant detector is both necessary and sufficient to solve CUP. Subsequently [2], the authors show that the same classes are not sufficient to solve FT-CUP.

In this section, we investigate the \( k \)-CO, \( k \)-SCO and \( k \)-OSR participant detectors with respect to the FT-CUP problem, assuming the lowest possible synchrony (i.e. the \( \Omega \) failure detector) necessary to solve consensus in known networks. In a nutshell, we show that provided the actual number of faults \( f \) is strictly lower than some constant \( k \)
known, (i) the \(k\)-CO participant detector is necessary to solve FT-CUP (Proposition 1), (ii) the \(k\)-SCO participant detector is sufficient to solve uniform FT-CUP assuming \(\Omega\) (Proposition 2), and (iii) the \(k\)-OSR participant detector is sufficient to solve uniform FT-CUP and necessary to solve FT-CUP assuming \(\Omega\) (Proposition 3).

3.1 \(k\)-CO Participant Detector is Necessary to Solve FT-CUP

Proposition 1 The \(k\)-CO participant detector is necessary to solve FT-CUP, in spite of \(f < k < n\) node crashes.

Proof: Assume by contradiction that the undirected knowledge connectivity graph \(G\) defined by the PD oracle is \((k - 1)\)-connected. Following observation 2 in Section 2.2, the removal of \(k - 1\) nodes may disconnect this undirected graph \(G\) into at least two components. From [1], connectivity of \(G\) is a necessary condition to solve CUP. So, to tolerate \(f < k\) node removals, \(\text{PD} \in k\)-CO.

3.2 \(k\)-SCO Participant Detector is Sufficient to Solve Uniform FT-CUP Assuming \(\Omega\)

Our approach to claim the main result of this section is constructive: we provide an algorithm (COLLECT) that enables the reuse of a previously known consensus algorithm assuming \(\Omega\).

3.2.1 The COLLECT Algorithm

Overview. The COLLECT algorithm (presented as Algorithm 1) provides nodes a partial view of the system participants. Each node eventually gets the maximal set of processes that it can reach. COLLECT considers that \(f < k\) processes may crash. When initiating the algorithm, a process \(p_i\) first queries its participant detector to obtain \(i\).PD; then \(p_i\) iteratively requests newly known processes to get knowledge improvement about the network, until no further knowledge can be acquired. Thus, COLLECT operates in rounds: in each round \(r > 0\), \(p_i\) contacts all nodes it didn‘t know about in round \(r - 1\) so that they increase \(p_i\)‘s knowledge about the network. At round 0, \(p_i\) only knows about itself. In our scheme, we assume that for each process \(p_i\), the participant detector \(i\).PD of \(p_i\) is queried exactly once. This can be implemented for example by caching the value of of the first result of \(i\).PD and returning that value in the subsequent calls. This property guaranties that the partial snapshot about the initial knowledge connectivity of the system is consistent for all nodes in the system, and defines a common knowledge connectivity graph \(G_{\text{di}} = (V, E)\).

Whenever \(\text{PD} \in k\)-SCO, COLLECT terminates and returns II. Otherwise, whenever \(\text{PD} \in k\)-OSR, the algorithm provides \(p_i\) all reachable nodes from its \(k\)-strongly-connected components plus reachable nodes from other components (which includes at least all nodes in the sink component).

On the example of Figure 1, COLLECT will return for \(p_i \in G_1\), a subset containing \(p_j \in \{G_1 \cup G_2 \cup G_3\}\); for \(p_i \in G_2\), a subset formed by \(p_j \in \{G_2 \cup G_3\}\); for \(p_i \in G_3\), a subset formed by \(p_j \in \{G_3\}\).

Variables. A process \(p_i\) manages the following local variables:
- \(i\).known: subset of processes known by \(p_i\) in the current round;
- \(i\).responded: subset of processes from which \(p_i\) has received a message;
- \(i\).previously known: previous set of processes known by \(p_i\) in the previous round;
- \(i\).wait: number of processes from which \(p_i\) is still waiting for a message.

Description. A process \(p_i\) starts the algorithm by executing the INIT phase (lines 13-15) in which \(p_i\) broadcast its knowledge (provided by the participant detector) about the system to every process in \(i\).PD, inviting the contacted processes to do the same in return. In this initial stage, \(p_i\) queries its participant detector (line 13) and sets \(i\).known to the returned list of participants (\(i\).PD). After that, it calls upon the Inquiry() procedure. Node \(p_i\) sends a message \(\text{VIEW}(i, i\).known\) to every known process \(p_j\) (lines 9-10) and updates some local variables. In particular, it sets \(i\).wait to the minimal number of correct nodes, i.e., the cardinality of its \(i\).known set minus the maximal number of crashes (\(f\)) (line 11).

In the IMPROVEMENT phase, upon receipt of a message \(\text{VIEW}(m\).initiator, m\).known\) from \(p_j\) to \(p_i\), two cases are presented.

1. \(m\).initiator \(\neq i\): this means that \(p_i\) have received a message from a remote node \(p_j\) querying its initial connectivity knowledge. Thus, \(p_i\) sends back to \(p_j\) its initial knowledge (line 28).
2. \(m\).initiator = \(i\): in this case, \(p_i\) received back a message carrying \(p_j\)‘s initial knowledge connectivity. Thus, in line 18, \(p_i\) improves its initial knowledge, extending \(i\).known with \(j\).PD. Then, \(p_i\) updates its local variables \(i\).responded and \(i\).wait (lines 19-20) accordingly. Afterwards, by testing the predicate \((i\).wait = 0\), \(p_i\) verifies whether it has received sufficiently many messages from all known correct nodes (line 21). If that is the case, \(p_i\) checks whether its current view has changed with respect to the previous one. Two situations can occur:

1. If \(i\).previously known \(\neq i\).known\), this means that \(p_i\) has gathered knowledge information from all known correct nodes. In this case, the algorithm terminates and \(p_i\) returns its \(i\).known set (line 23).
Algorithm 1 COLLECT()

constant:
(1) \( f \): int \( // \) upper bound on the number of crashes
variables:
(2) \( i \).previously_known : set of nodes
(3) \( i \).known : set of nodes
(4) \( i \).responded : set of nodes
(5) \( i \).wait : int
message:
(6) VIEW message:
(7) initiator : node
(8) known : set of nodes

procedure:
Inquiry():
(9) for \( j \) in \( i \).known \( \setminus \) \( i \).previously_known do
(10) send VIEW (\( i \), \( i \).known) to \( p_j \); end do
(11) \( i \).wait = \( |i \).known \( \setminus \) \( i \).responded\( | - f \);
(12) \( i \).previously_known = \( i \).known;

** Initiator Only **
INIT:
(13) \( i \).known = \( i \).PD;
(14) \( i \).responded = \( i \).previously_known = \{\};
(15) call upon Inquiry();

** All Nodes **
IMPROVEMENT:
(16) upon receipt of VIEW(m.initiator, m.known) from
\( p_i \) to \( p_j \):
(17) if \( i \) == m.initiator then
(18) \( i \).known = \( i \).known \( \cup \) m.known;
(19) \( i \).responded = \( i \).responded \( \cup \) \{\};
(20) \( i \).wait = \( i \).wait \(- f \);
(21) \( i \).wait = 0 then
(22) if \( i \).previously_known == \( i \).known then
(23) return (\( i \).known);
(24) else
(25) call upon Inquiry(); end if
(26) \( i \).responded = \( i \).previously_known = \{\};
(27) else
(28) send VIEW(m.initiator, i.PD) to \( p_j \);
(29) end if

(2) If \( i \).previously_known \( \neq \) \( i \).known, this means that
\( p_i \) has discovered new nodes. So, it will start a new round
to improve knowledge information about the new nodes be-
longing to \( i \).known \( \setminus \) \( i \).previously_known. So, \( p_i \) calls the
Inquiry() procedure to send a message VIEW(\( i \), \( i \).known) to
every new node recently discovered (line 9-10). After that,
\( p_i \) updates \( i \).wait accordingly, excluding those having al-
ready responded and crashed. Finally, \( i \).previously_known
receives the contents of the most recent \( i \).known set.

Lemma 1 The algorithm COLLECT proceeds execution by rounds,
and the number of rounds is finite.

Proof: Inspecting the algorithm reveals that it proceeds exec-
ution by rounds. Each round \( r \) is started when the Inquiry()
procedure is called upon by the initiator. In the beginning,
the algorithm starts round \( r = 1 \), by executing lines 13-15
and calling upon Inquiry(). Afterwards, each round \( r > 1 \)
is started whenever new informations about processes in
the system are gathered by \( p_i \) in round \( (r - 1) \), thus sat-
ifying the condition \( i \).known \( \neq \) \( i \).previously_known in
line 24. The algorithm proceeds by executing sequential
rounds until it eventually terminates, which happens when-
ever \( i \).known \( \neq \) \( i \).previously_known is satisfied, mean-
ing that the improvement on \( p_i \)’s knowledge has finished
(line 23).

No assume that the number of rounds in infinite. This
implies that \( p_i \) receives new knowledge about new pro-
cesses infinitely often. As a consequence, the number of
processes in the system is infinite. Thus, a contradiction. \( \Box \)

Lemma 2 Starting by round \( r = 1 \), in each round \( r \) of al-
gorithm COLLECT, \( i \).known is augmented with reachable
nodes whose distance from \( p_i \) is \( r \).

Proof: To discover the set of reachable processes, the
algorithm COLLECT realizes a sort of breadth-first search
in the graph \( G_d \). Let the initiator \( p_i \), be the root of the tree
established by this search. The rounds correspond to the
levels of the tree. If \( p_j \) is first discovered by \( p_i \) in round
\( r \), then \( d(p_j, p_i) = r \). This means that \( p_j \) is reached by
the breadth-first search in level \( r \). Denote \( N_r(p_i) \) the set
of all nodes reached by the breadth-first search until level
\( r \). Let \( i \).known be the set of known nodes in round \( r \). So,
\( i \).known = \( N_r(p_i) \). Let us proceed the proof by induction
on \( r \).

Basis: In round \( r = 1 \) (level 1 of the tree), \( p_i \) attributes
its list of adjacent nodes to \( i \).known, which corresponds to
the list of participants returned by \( i \).PD (line 13). So, if \( p_j \in
i \).PD, \( d(p_j, p_i) = 1 \). Let \( N_{(1)}(p_i) \) be the set of adjacent
nodes. So, \( i \).known = \( N_{(1)}(p_i) \).

Induction: Suppose the Lemma holds for level \( < r \) of
the tree. By Lemma 1, round \( r \) starts whenever infor-
mations about new nodes in the system are gathered by
Note that, on the execution of this investigation procedure condition (i \text{.known} \neq i \text{.previously\_known}) in line 24. Let \( p_j \) be a node in (i \text{.known} \setminus i \text{.previously\_known}). Thus, \( p_j \) has been discovered by \( p_i \) in round \( (r - 1) \). This means that \( p_j \) is reached by the breadth-first search in level \((r - 1)\).

Starting round \( r \), \( p_i \) inquires \( p_j \) for sending its view of known processes (lines 9-10). After that, still in round \( r \), node \( p_j \) will reply, passing back to \( p_i \) the list of participants returned by its participant detector \( j \text{.PD} \) (line 28). Let \( p_l \in j \text{.PD} \) and \( p_l \notin i \text{.known} \). This means that, in its probing for discovering new processes, \( p_l \) has not met \( p_i \) (round \( < r \)) otherwise, by the inductive hypothesis \( p_i \) would be in \( i \text{.known} \). Thus, \( d(p_i, p_l) > (r - 1) \).

Upon reception of message \( \text{VIEW} \) from \( p_j \), \( i \text{.known} \) is updated with \( j \text{.PD} \) (line 18). By the inductive hypothesis, in round \((r - 1)\), \( i \text{.known} = N_{(r-1)}(p_i) \). Thus, in round \( r \), \( i \text{.known} \) contains \( N_{(r-1)}(p_i) \), extended with every new process discovered by \( p_i \) in round \( r \) (including \( p_i \)). So, in round \( r \), \( i \text{.known} = N_r(p_i) \). By definition, \( (p_i, p_l) \in G_{di} \) defined by \( j \text{.PD} \), \( d(p_j, p_i) = 1 \). By the inductive hypothesis, \( d(p_i, p_j) = r - 1 \). Thus, \( d(p_i, p_l) = r \).

Thus, in round \( r \), \( i \text{.known} \) is augmented with nodes whose distance from \( p_i \) is \( r \). \( \square \)

**Lemma 3** Consider a \( k \text{-OSR} \) participant detector. Let \( f < k \) be the number of nodes that may crash. Algorithm COLLECT (1) executed by each node (i.e. having each node being an initiator) satisfies the following properties:

- **Termination:** every node \( p_i \) terminates execution and returns a list of known nodes (processes with whom \( p_i \) can communicate);
- **Safety:** algorithm COLLECT returns the maximal set of correct processes reachable from \( p_i \).

**Proof:**

**Termination.** Let us proceed our proof by induction on \( r \). In round \( r = 1 \), at beginning of the execution, \( i \text{.known} \) receives the list from \( i \text{.PD} \) (line 13). So, \( i \text{.known} \) is initially composed by processes with whom it can communicate. Going on the round, at line 15, \( p_i \) calls upon the \( \text{Inquiry}() \) procedure, so that it will send a \( \text{VIEW}() \) message to every one of these \( i \text{.known} \) processes, excluded those in \( i \text{.previously\_known} \) (which in round \( r = 1 \) is empty) (lines 9-12). By Menger’s Theorem, there are at least \( k \) nodes in each one of the \( m \) components of \( G_{di} \). Since \( f < k \), there are at least \( 1 \) correct node in each one of these components. So, \( p_i \) will receive at least \( (|i \text{.known}| - f) \geq 1 \) responses for its inquiry (line 16). This number coincides to the initial value of the \( i \text{.wait} \) variable (set up in line 11) and thus, due to its decay when a reply arrives (line 20), eventually condition \((i \text{.wait} == 0)\) will be satisfied (see line 21). Note that, on the execution of this investigation procedure – characterized by the sent and reception of \( \text{VIEW}() \) messages – \( p_i \) could enlarge its knowledge about processes in the system, resulting in the update of its \( i \text{.known} \) set (line 18). Note also that \( p_i \)’s previous knowledge is stored in the \( i \text{.previously\_known} \) set (see line 12). Whenever the condition \((i \text{.wait} == 0)\) is verified, two case are possible:

(i) \( i \text{.known} = i \text{.previously\_known} \). This means that correct processes in \( i \text{.known} \) share the same view. In this case, the algorithm terminates by returning the gathered \( i \text{.known} \) view (line 23).

(ii) \( i \text{.known} \neq i \text{.previously\_known} \). This means that \( p_i \) has enlarged its knowledge. In this case, it will inquiry for the view of these new processes, calling upon \( \text{Inquiry()} \) and starting a new round \( r + 1 \) (line 25). Suppose these executing conditions hold in rounds \( < r \). Eventually, in round \( r \), since the set of processes in the system \( (\Pi) \) is finite, no new process is going to be discovered by \( p_i \) in line 18. Thus, condition (i) \((i \text{.known} = i \text{.previously\_known})\) will be satisfied and the algorithm terminates.

**Safety.** Let us first make some useful remarks. Let \( G_{di} = (V, E) \) be the knowledge graph defined by \( k \text{-OSR} \) and decomposed into its \( m \) \( k \)-strongly-connected components. Let \( G = G_1 \cup G_2 \cup ... \cup G_m \) be such a decomposition. Remember that there is exactly one sink component in \( G_{di} \). Note also that \( V = \Pi \).

Consider two nodes \( p_i \) and \( p_j \) in \( V \). Two cases are possible. (i) If \( p_i \) and \( p_j \) are in the same component \( G_i \), since each one of the \( G_{di} \) components is \( k \)-strongly-connected, there is at least \( k \) node-disjoint paths between any two nodes in \( G_i \); (ii) If \( p_i \in G_i \) and \( p_j \in G_j \), \( G_i \neq G_j \) (the nodes are in distinct components), suppose that \( p_j \) is reachable from \( p_i \) (\( p_i \rightarrow p_j \)), from the property (3) of the graph \( G_{di} \) generated by \( k \text{-OSR} \), there are \( k \)-disjoint paths from \( G_i \) to \( G_j \). So, there is at least \( k \) node-disjoint paths from node \( p_i \) to \( p_j \) in \( G_{di} \). From the graph connectivity (see the observation 2 in Section 2.2), removing \( (k - 1) \) nodes leaves at least one path between any pair of nodes \((p_i, p_j)\) in each \( k \)-strongly-connect component. Thus, in situation (i), the graph remains strongly connected, meaning that there is at least one path of correct nodes from every pair of nodes. In situation (ii), there is at least one path from \( p_i \) to \( p_j \) composed of correct nodes.

Our claim is that algorithm COLLECT returns to \( p_i \) the maximal set of correct processes reachable from \( p_i \). This set is stored in \( i \text{.known} \). Let us proceed our proof by induction on the number of rounds and demonstrate that, in round \( r \), \( i \text{.known} \) contains all reachable processes from \( p_i \) through a path of length at most \( r \).

In round \( r = 1 \), \( i \text{.known} \) contains all neighbor nodes returned by its participant detector \( i \text{.PD} \). From Lemma 2, \( d(p_i, p_j) = 1 \).

Suppose the claim is valid for round \( < r \). Let \( p_l \) be a
node such that \( d(p_i, p_l) = r \). In this case, from the state-
ments above (situations (i) and (ii)), there is at least one path from \( p_i \) to \( p_l \) composed of correct nodes. Let \( p_j \) be the predecessor of \( p_l \) in this path. Thus, \( p_j \) belongs to \( j.PD \). Evidently \( d(p_j, p_l) = (r-1) \); otherwise, \( d(p_i, p_l) \neq r \). By the inductive step, \( i.\text{known} \) contains all those correct nodes that are exactly \((r-1)\) edges away from \( p_i \). Since \( d(p_j, p_l) = (r-1) \), \( p_j \) has been discovered by \( p_i \) in round \((r-1)\) (Lemma 2).

Round \( r \) starts whenever informations about new nodes in the system are gathered by \( p_i \) in round \((r-1)\), (see Lemma 1). Thus, in round \((r-1), p_j \in (i.\text{known} \setminus i.\text{previously known}) \). In round \( r \), at the beginning, \( p_i \) will inquiry all new nodes (including \( p_j \)) to send its view of known processes (lines 9-10). After that, still in round \( r \), node \( p_j \) will reply, by passing back to \( p_i \) its list of participants returned by its participant detector \( j.PD \) (line 28). Upon reception of message \( \text{view} \) from \( p_j \), \( i.\text{known} \) is updated with \( j.PD \) (18). By the inductive step, in round \((r-1), i.\text{known} \) contains all processes reachable from \( p_i \) through a path of length at most \((r-1)\). Thus, in round \( r, i.\text{known} \) is extended with every new node discovered by \( p_i \) in round \( r \) (thus including \( p_j \)). So, in round \( r, i.\text{known} \) contains all correct nodes reachable from \( p_i \) through a path of length at most \( r \).

These Corollaries below follow directly from Lemma 3.

**Corollary 1** Consider a \( k\)-\( \text{SCO} \) participant detector. Let \( f < k < n \) be the number of nodes that may crash. Algorithm \( \text{COLLECT} \) (1) executed by each node results in every correct node getting upon termination the knowledge of the composition of its \( k\)-strongly-connected component plus reachable nodes from other components (which includes at least all nodes in the sink component).

**Corollary 2** Consider a \( k\)-\( \text{OSR} \) participant detector. Let \( f < k < n \) be the number of nodes that may crash. Processes within the same \( k\)-strongly-connected component have the same view of reachable processes (\( \text{known set} \)) after executing algorithm \( \text{COLLECT} \) (1).

**Corollary 3** Consider a \( k\)-\( \text{SCO} \) participant detector. Let \( f < k < n \) be the number of nodes that may crash. Algorithm \( \text{COLLECT} \) (1) executed by each node results in every correct node getting upon termination the knowledge of \( \Pi \).

**Proposition 2** The \( k\)-\( \text{SCO} \) participant detector is sufficient to solve uniform \( \text{FT-CUP} \), in spite of \( f < k < n \) node crashes, assuming \( \Omega \).

**Proof:** Sufficient: The \( \text{COLLECT} \) algorithm provides each process \( p_i \) with the set \( \Pi \) (see Corollary 3), in spite of \( f < k \) crashes. Then, previous indulgent algorithms aiming for solving classical consensus, which are based on a priori knowledge about \( \Pi \), can be used [3, 8]. In particular, if \( f < k/2 \), and \( k < n \), it is possible to solve \( \text{FT-CUP} \) as well uniform \( \text{FT-CUP} \) in a system enriched with both: a \( k\)-\( \text{SCO} \) participant detector and a \( \Omega \) failure detector. \( \square \)

### 3.3 \( k\)-\( \text{OSR} \) Participant Detector is Sufficient and Necessary to Solve \( \text{FT-CUP} \) Assuming \( \Omega \)

Our approach for the main result of this section is also constructive. The \( \text{CONSENSUS} \) algorithm that we provide builds upon the previously presented \( \text{COLLECT} \) algorithm and a second algorithm (\( \text{Sink} \)) that determines whether a node is in the single \( k\)-strongly connected sink component of the knowledge connectivity graph.

#### 3.3.1 The \( \text{Sink} \) Algorithm

The \( \text{Sink} \) algorithm (presented as Algorithm 2) determines if a node belongs to the sink component. \( \text{Sink} \) makes use of the \( \text{COLLECT} \) algorithm that provides nodes with a partial view of the system composition. Now, in the sink component, nodes have the same view of the system (i.e. the same set of known nodes), whereas in the other components, nodes have strictly more knowledge than in the sink.

The algorithm is composed of two phases. In the \( \text{Init} \) phase, processes broadcast their knowledge about the composition of the system (which is an approximation of \( \Pi \)), while in the \( \text{Verification} \) phase, processes determine whether they belong to the sink component or not.

**Variables.** A process \( p_i \) maintains the following local variables:

- \( i.\text{known} \): subset of processes currently known by \( p_i \);
- \( i.\text{responded} \): subset of processes from which \( p_i \) has received an \( \text{ack} \) message;
- \( i.\text{in}_\text{the}_\text{sink} \): predicate indicating whether \( p_i \) belongs to the sink component.

**Description.** A process \( p_i \) starts the algorithm by executing the \( \text{Init} \) procedure (lines 9-12). First, \( p_i \) runs the \( \text{Collect} \) algorithm to get the partial list of nodes composing the system. This list is stored in \( i.\text{known} \) (line 9). Afterwards, \( p_i \) sends a \( \text{Request}(i.\text{known}) \) message to every process \( p_j \) in this set (lines 11-12). Upon receipt of a \( \text{Request}(m.\text{known}) \) message from \( p_j \), process \( p_i \) tests if its own \( i.\text{known} \) set is equal to the message’s \( m.\text{known} \) set. In case of equality, this means that \( p_j \) belongs to the same component of \( p_j \) (Corollary 2). So, \( p_i \) sends back an \( \text{ack} \) response to \( p_j \) (line 15). Otherwise, \( p_i \) sends back a \( \text{nack} \) response (line 17).

Upon receipt of a \( \text{Response}(\text{ack/nack}) \) message from \( p_j \), process \( p_i \) determines whether it is in the sink component or not. If \( p_j \) responded \( \text{nack} \), this means that \( p_i \) has identified processes (including \( p_j \)) belonging to other
components. So, \( p_i \) cannot be in the sink and it terminates execution returning \text{false} for the \text{i.in\_the\_sink} predicate (line 25-26).

**Algorithm 2 SINK ()**

<table>
<thead>
<tr>
<th>constants:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( f ): upper bound on the number of crashes</td>
</tr>
<tr>
<td>(2) ( i.known ): set of nodes</td>
</tr>
<tr>
<td>(3) ( i.in_the_sink ): boolean</td>
</tr>
<tr>
<td>(4) ( i.responded ): set of nodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5) ( \text{REQUEST} ) message:</td>
</tr>
<tr>
<td>(6) ( \text{known} ): set of nodes</td>
</tr>
<tr>
<td>(7) ( \text{RESPONSE} ) message:</td>
</tr>
<tr>
<td>(8) ( \text{ack/nack} ): boolean</td>
</tr>
</tbody>
</table>

**All Nodes**

<table>
<thead>
<tr>
<th>INIT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(9) ( i.known = \text{COLLECT}(); )</td>
</tr>
<tr>
<td>(10) ( i.responded = { }; )</td>
</tr>
<tr>
<td>(11) for each ( j ) in ( i.known ) do</td>
</tr>
<tr>
<td>(12) send ( \text{REQUEST}(i.known) ) to ( p_j ); endfor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VERIFICATION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13) upon receipt of ( \text{REQUEST}(m.known) ) from ( p_j )</td>
</tr>
<tr>
<td>(14) if ( m.known == i.known ) then</td>
</tr>
<tr>
<td>(15) send ( \text{RESPONSE}(\text{ack}) ) to ( p_j );</td>
</tr>
<tr>
<td>(16) else</td>
</tr>
<tr>
<td>(17) send ( \text{RESPONSE}(\text{nack}) ) to ( p_j ); endif</td>
</tr>
</tbody>
</table>

| (18) upon receipt of \( \text{RESPONSE}(m) \) from \( p_j \) |
| (19) if \( m.ack \) then |
| (20) \( i.responded = i.responded \cup \{ j \} \); |
| (21) if \( |i.responded| > |i.known| - f \) then |
| (22) \( i.in\_the\_sink = \text{true} \); |
| (23) return \( (i.in\_the\_sink, i.known) \); endif |
| (24) else |
| (25) \( i.in\_the\_sink = \text{false} \); |
| (26) return \( (i.in\_the\_sink, i.known) \); |
| (27) endif |

If \( p_j \) responds \( \text{ack} \), this means that \( p_j \) has the same view of \( p_i \) about reachable processes in the system. Moreover, if \( p_j \) receives \( \text{ack} \) messages from every correct process in its view, \( p_j \) can conclude that it is in the sink component. So, when receiving an \( \text{ack} \) message, \( p_j \) updates its local variable \( i.responded \) to take into account \( p_j \)'s response (line 20) and tests the condition \( |i.responded| > |i.known| - f \) in order to know if it has received responses from every correct process (line 21). Whenever this condition becomes \text{true}, \( p_i \) is sure that it belongs to the sink component and thus it can terminate execution, returning \text{true} for the \text{i.in\_the\_sink} predicate (lines 22-23).

**Lemma 4** Consider a \( k\)-OSR participant detector. Let \( f < k < n \) be the number of nodes that may crash. Algorithm \text{SINK} (2) executed by each node satisfy the following properties:

- **Termination**: every node \( p_i \) terminates execution by deciding whether it belongs to the sink component (true) or not (false);
- **Safety**: a node \( p_i \) is in the unique \( k\)-strongly connected sink component iff algorithm \text{SINK} returns true.

**Proof:** Termination. At the beginning of execution, node \( p_i \) sends a \text{REQUEST} message to all processes in its local view \( (i.known) \) (lines 11-12). Since at most \( f < k \) processes can crash, \( p_i \) will receive at least \( s = (|i.known| - f) \) responses in line 13. Since \( G_{di} \) is \( k\)-strongly connected, \( |i.known| \geq k \), thus \( s \geq k - f \geq 1 \). If one of these responses equals \( \text{nack} \), the algorithm terminates, by returning \text{false} (lines 25-26). If a sufficient number \( (\geq s) \) of \( \text{ack} \) responses is received (line 21), the algorithm terminates by returning \text{true} (lines 22-23). Lines 23 and 26 are the only points where the algorithm terminates. Thus \text{true} or \text{false} are the only possible returns.

Safety. (i) Let us first prove that if node \( p_i \) is in the unique \( k\)-strongly connected sink component then algorithm \text{SINK} returns true. From Lemma 3, the \text{COLLECT} algorithm returns a list of all nodes reachable from \( p_i \) in \( G_{di} \). Consequently, nodes in the unique \( k\)-strongly connected sink will have the same view of the system (Corollary 2) and the execution of line 9 returns the same \( i.known \) set to all nodes in the sink. In this case, every node \( p_j \) in view \( i.known \) which executes line 13 will respond \text{ack} to \( p_i \)'s request (line 15). Thus, the condition in line 24 will never be satisfied. Moreover, since there are at least \( s = (|i.known| - f) \) correct processes in the system, at least a number of \( s \) responses will be received by \( p_i \). Thus, condition in line 21 will eventually be satisfied and the algorithm terminates returning \text{true} (lines 22-23).

(ii) Let us now prove that if algorithm \text{SINK} returns \text{true} then node \( p_i \) is in the unique \( k\)-strongly connected sink component. Assume by contradiction that \( p_i \) does not belong to the unique sink of \( G_{di} \). If that is the case, \( i.known \) is composed by processes belonging to other components than \( p_i \)'s (Corollary 1). By the connectivity of the graph, there are at least \( k \) nodes in each one of the \( m \) components in \( G_{di} \). Since \( f < k \), there are at least \( 1 \) correct node in each one of these components. So, \( p_i \) will receive in line 18 at least \( 1 \) \text{nack} response from a process \( p_l \) belonging to other components than \( p_i \)'s. Moreover, \( p_i \) will never receive, at line 19, \( s \geq (|i.known| - f) \) of \( \text{ack} \) responses, since at least \( 1 \) of those responses from a correct process will be \( \text{nack} \). Thus the condition in line 21 will never be satisfied. So,
3.3.2 The CONSENSUS Algorithm

The CONSENSUS protocol is presented as Algorithm 3. In the initial phase, every node runs SINK (Algorithm 2) to get a partial view of the system and decide whether or not it belongs to the $k$-strongly connected sink component. Depending on whether the node belongs or not to the sink, two behaviors are possible.

For the nodes belonging to the sink, an AGREEMENT phase is launched in order to reach a consensus on some value. By construction, all nodes in the sink component share the same $i$.known set, so using $\Omega$ is sufficient to solve consensus as soon as there are at least $2k + 1$ nodes in the sink component. The other nodes (in the remaining $k$-strongly connected components) do not participate to this consensus. They launch a REQUEST phase to ask for and collect the value decided by the sink members. This is done by sending request messages to known processes and waiting for responses. Since at least one member from the sink is correct, at least one member will respond the decided value when it is decided.

**Proposition 3** The $k$-OSR participant detector is sufficient to solve uniform FT-CUP and necessary to solve FT-CUP, in spite of $f < k < n$ crashes, assuming $\Omega$.

**Proof**: Sufficient: Algorithm 3 solves uniform FT-CUP with PD $\in k$-OSR, assuming $\Omega$. The following statements proves this claim.

Validity. This property trivially follows from the fact that a decided value is a value proposed by nodes in the sink component (line 16).

Termination. To prove that every correct process decides, we must prove that they finish by executing lines 21 or 32 of the algorithm. On the execution of the main decision task, we can distinguish two types of behavior: (i) that one from the nodes belonging to the sink and (ii) that from the nodes not in the sink component. In case (i), nodes in the sink will call upon a classical indulgent protocol which solves consensus (line 16). From the termination property of this algorithm, a decision is eventually attained and then line 17 is executed by every node in the sink. Thus, after executing lines 18-21, a decision is returned to the application (line 21). In case (ii), nodes not in the sink will send a message requesting for the decision to all the nodes in their $i$.known set returned by the COLLECT procedure executed in the SINK algorithm (lines 27-28). From Corollary 1, every node in the sink belongs to $i$.known. Thus, after receiving the request message in task $T_2$ from a node $p_j$ not in the sink (line 22), a node in the sink will pass back the

\[
\begin{align*}
\text{Algorithm 3 CONSENSUS} \\
\text{constant:} & \quad (1) \quad f: \text{upper bound on the number of crashes} \\
\text{input:} & \quad (2) \quad i$.initial$: \text{value} \\
\text{variable:} & \quad (3) \quad i$.in\_the\_sink$: \text{boolean} \\
& \quad (4) \quad i$.known$: \text{set of nodes} \\
& \quad (5) \quad i$.decision$: \text{value} \\
& \quad (6) \quad i$.asked$: \text{set of nodes} \\
\text{message:} & \quad (7) \quad \text{REQUEST message} \\
& \quad (8) \quad \text{RESPONSE message} \\
& \quad (9) \quad \text{decision}: \text{value} \\
\end{align*}
\]

**All Nodes**

\text{request}: \\
(10) \quad i$.asked$ = \{\}; \quad i$.decision$ = \perp; \\
(11) \quad (i$.in\_the\_sink$, i$.known$) = \text{SINK}(); \\
if i$.in\_the\_sink$ \text{then} \\
(13) \quad \text{fork AGREEMENT} \\
(14) \else \text{fork REQUEST end if} \\
**Node In Sink**

\text{AGREEMENT}: \{ \text{make use of classical consensus} \} \\
(16) \quad \text{Consensus.propose}(i$.initial)$ \\
(17) \quad \text{upon Consensus.decide}(v)$: \\
(18) \quad i$.decision$ = v; \\
(19) \quad \text{for every } j \text{ in } i$.asked$ \text{do} \\
(20) \quad \text{send RESPONSE (i$.decision$) to } p_j; \text{end for} \\
(21) \quad \text{return (i$.decision$);} \\
\text{task T2:} \{ \text{Decision Dissemination Task} \} \\
(22) \quad \text{upon receipt of REQUEST()} \text{ from } p_j$; \\
(23) \quad \text{if i$.decision$ \neq \perp$ then} \\
(24) \quad \text{send RESPONSE (i$.decision$) to } p_j; \\
(25) \else \text{end if} \\
(26) \quad i$.asked$ = i$.asked$ \cup \{j\}; \\
**Node Not In Sink**

\text{request}: \\
(27) \quad \text{for every } j \text{ in } i$.known$ \text{do} \\
(28) \quad \text{send REQUEST () to } j; \\
(29) \quad \text{upon receipt of RESPONSE (v) from } j$; \\
(30) \quad \text{if i$.decision$ = \perp$ then} \\
(31) \quad i$.decision$ = v; \\
(32) \quad \text{return (i$.decision$);} \text{end if}
than following scenarios are possible: (i) either there exists less composed into its $k$ disconnect the graph into at least two components. Since sink. In the first scenario, the crash of $k$ or (ii) the decomposition of $G$ for solving CUP [1]. Assume by contradiction that there is every node in the sink component will receive the same value $v$ in line 17 for the decision. So, every one of these nodes will diffuse the same value $v$, immediately after taken the decision on the execution of lines 19-20, or in the “decision dissemination task” T2 (lines 22-26).

Uniform Agreement. The guarantee that no two processes decide differently comes directly from the uniform agreement property of the underlying indulgent consensus. Thus, every node in the sink component will receive the same value $v$ in line 17 for the decision. So, every one of these nodes will diffuse the same value $v$, immediately after taken the decision on the execution of lines 19-20, or in the “decision dissemination task” T2 (lines 22-26).

Necessary: Let us give a sketch of the proof which is based on the same arguments to prove the necessity of OSR for solving CUP [1]. Assume by contradiction that there is an algorithm $A$ which solves FT-CUP with a PD $\not\in k$-OSR. Let $G_{di}$ be the knowledge graph induced by PD decomposed into its $k$-strongly connected components. The following scenarios are possible: (i) either there exists less than $k$ node-disjoint paths between two components of $G_{di}$; or (ii) the decomposition of $G_{di}$ originates more than one sink. In the first scenario, the crash of $k - 1$ nodes may disconnect the graph into at least two components. Since connectivity is a necessary condition to solve CUP [1], we reach a contradiction. In the second scenario, let $G_1$ and $G_2$ be two of those sinks. Assume that all processes in $G_1$ have input value equal to $v$ and that all processes in $G_2$ have input value equal to $w$, $v \neq w$. By the termination property of consensus, processes in $G_1$ decide at time $t_1$ and processes in $G_2$ decide at time $t_2$. We can delay the reception of any messages from processes in other components to both $G_1$ and $G_2$ to a time $t > \max\{t_1, t_2\}$. Since processes in the sinks are unaware about the existence of other processes, by the validity property of consensus, processes in $G_1$ decide for the value $v$ and processes in $G_2$ decide for the value $w$, violating the agreement and reaching thus a contradiction.

4 Discussion

In this paper, we investigated the trade-off between knowledge about the system and synchrony assumptions to enable consensus in fault-prone unknown systems. It turns out that if knowledge connectivity is $k$-OSR, then consensus can be solved assuming minimal synchrony assumptions. Our approach is constructive, and an interesting side effect of our design is that the uniform version of the consensus can be solved as well, with no particular effort. This complements nicely previous studies that showed that complete synchrony was needed whenever only minimal knowledge connectivity (OSR) was available. Interestingly enough, the same previous solution did not enable uniform consensus.

Our works leads to several interesting open questions:
1. It is still unclear whether the performance of the consensus algorithm is impacted by the knowledge connectivity of the system and by the synchrony assumptions. Would it more interesting to boost connectivity knowledge or synchrony to make decisions faster?
2. All known solutions to the FT-CUP problem have the consensus value chosen from the initial value of a particular set of nodes (the ones in the “sink” component). What would be the knowledge connectivity requirement (and the associated synchrony assumption) to have all initial values taken into account? (Besides the obvious SCO and k-SCO classes.)

References