Knowledge Connectivity Requirements for Solving Byzantine Consensus with Unknown Participants

Eduardo Adilio Pelinson Alchieri, Alysson Bessani, Fabíola Greve, Joni da Silva Fraga

Abstract—Consensus is a fundamental building block to solve many practical problems that appear on reliable distributed systems. In spite of the fact that consensus is being widely studied in the context of standard networks, few studies have been conducted in order to solve it in dynamic and self-organizing systems characterized by unknown networks. While in a standard network the set of participants is static and known, in an unknown network, such set and number of participants are previously unknown. This work studies the problem of *Byzantine Fault-Tolerant Consensus with Unknown Participants*, namely BFT-CUP. This new problem aims at solving consensus in unknown networks with the additional requirement that participants in the system may behave maliciously. It presents the necessary and sufficient knowledge connectivity conditions in order to solve BFT-CUP under minimal synchrony requirements. In this way, it proposes algorithms that are shown to be optimal in terms of synchrony and knowledge connectivity among participants in the system.

Index Terms—Distributed Agreement, Consensus with Unknown Participants, Byzantine Fault Tolerance, Self-organizing Systems.

1 Introduction

The *consensus* problem [1], [2], [3], [4], [5], and more generally *agreement* problems, form the basis for most solutions related to the development of reliable distributed systems [6], [7]. Through these protocols, participants are able to coordinate their actions in order to maintain state consistency and ensure system progress. Consensus has been extensively studied in standard networks, where the set of processes involved in a particular computation is static and known by all participants in the system. Nonetheless, even in these environments, the consensus problem has no deterministic solution in presence of one single process crash, when entities behave asynchronously [3]. Due to this limitation, usually some synchrony need to be assumed in the system [1], [8].

In self-organizing systems, such as wireless mobile ad-hoc networks, sensor networks and unstructured peer to peer networks (P2P), solving consensus is even more difficult. In these environments, initial complete knowledge about the participants in the system is a strong assumption since the system composition changes frequently. These environments define indeed a new model of dynamic distributed systems which has essential differences regarding the standard static networks. Consequently, it brings new challenges to the specification and resolution of problems.

- *E. A. P. Alchieri is with Department of Computer Science, University of Brasília, Brazil.*
- *A. Bessani is with LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal.*
- *F. Greve is with Computer Science Department (DCC), Federal University of Bahia, Brazil.*
- *J. Fraga are with Department of Automation and Systems (DAS), Federal University of Santa Catarina, Brazil.*

- *This work is supported by CNPq (Brazil), FCT (Portugal) and European Commission through projects FREESTORE (CNPq 457272/2014-7), SUPERCLOUD (H2020-ICT 643964), LASIGE (PEst-OE/EEI/UI0408/2014), and IRCoC (PTDC/EEI-SCR/6970/2014).*

Most of the studies about consensus are not suitable for these systems because they assume a static and known set of participants (e.g., [1], [2], [4], [9], [10], [11]). Some notably exceptions are the works of Cavin *et al.* [12], [13] and Greve *et al.* [14], [15] for the crash failure model and the work of Alchieri *et al.* [16] for the Byzantine failure model. These works identify necessary and sufficient knowledge connectivity requirements to solve consensus when the set of participants is unknown in the system. The work presented herein extends these previous results providing novel algorithms and knowledge connectivity conditions.

Related Work. Cavin *et al.* [12] defined the CUP problem (*consensus with unknown participants*) to solve consensus in a failure-free asynchronous network with unknown participants. With this aim, the *participant detector* abstraction (namely, PD) has been defined to provide processes with an initial knowledge about the system membership. The work establishes the necessary and sufficient knowledge connectivity conditions able to solve CUP, which are represented by the *One Sink Reducibility* participant detector (namely, OSR). In a subsequent study [13], the same authors extend their results to a crash-prone model and provide a solution to FT-CUP (*Fault-Tolerant CUP*). They show that to solve FT-CUP with the minimal requirements regarding knowledge connectivity (represented by the OSR PD), it is necessary to enrich the system with the *Perfect* ($\mathcal{P}$) failure detector [1].

Greve and Tixeuil [14] go one step further and show that there is in fact a trade-off between knowledge connectivity and synchrony for consensus in fault-prone unknown networks. They provide an alternative solution for FT-CUP which requires minimal synchrony assumptions; indeed, the same assumptions already identified to solve consensus in a standard environment, which are represented by *Eventually Strong* ($\diamond\mathcal{S}$) failure detectors [1]. They prove that the k-OSR PD [14] unify the necessary and sufficient requirements to

"This paper has been published in IEEE Transactions on Dependable and Secure Computing (Volume:PP , Issue: 99) in March 2016 and is available at IEEE via: http://dx.doi.org/10.1109/TDSC.2016.2548460"

Table 1
SOLUTIONS for the CONSENSUS WITH UNKNOWN PARTICIPANTS.

Work	Failure Model	PD	Sink Size	Connectivity	Synchrony
CUP [12]	no failures	OSR	1	OSR	asynchronous
FT-CUP [13]	crash	OSR	1	safe crash pattern	asynchronous + $\mathcal{P}$
FT-CUP [14], [15]	crash	k-OSR	$2f+1$	$f+1$ node-disjoint paths	asynchronous + $\diamondsuit\mathcal{S}$
BFT-CUP [16]	Byzantine	k-OSR	$3f+1$	$2f+1$ node-disjoint paths	as the underlying consensus
BFT-CUP (this paper)	Byzantine	k-OSR	$2f+1$ correct	safe Byzantine failure pattern	as the underlying consensus

solve uniform FT-CUP, assuming $\diamondsuit\mathcal{S}$.

In an earlier version of this work [16], we studied the FT-CUP problem under Byzantine failures [4] (*Byzantine FT-CUP* or BFT-CUP) and identified conditions for solving it in an asynchronous system. By using a path of reductions similar to [14], we provided a solution to BFT-CUP in a system extended with the same participant detector: k-OSR.

The solutions presented in all these works start by using the information given by the PD that forms a "knowledge" connectivity graph, in which an edge between participants i and j represents the fact that i initially knows j. Using such initial knowledge, each process tries to expand the set of processes it knows, by exchanging the knowledge about the system with other participants until a set of participants that share exactly the same view of the system is identified. These participants form what we call the *sink* of the knowledge graph, and run any standard consensus protocol (for know networks), sending the decided value to other *non-sink* participants by request.

Table 1 summarizes the requirements for solving CUP, FT-CUP and BFT-CUP in terms of number of participants in the sink, connectivity and synchrony. To the best of our knowledge, these are the only works to study knowledge connectivity conditions necessary for solving consensus in a system with unknown participants, considering the message-passing model.

Besides the works summarized in Table 1, there are works studying the FT-CUP problem in a shared memory model [17], or other distributed computing problems such as leader election [18], resource allocation [19] and failure detection [20] when participants are unknown. None of these works consider Byzantine failures.

Contributions. This paper extends these previous results, notably [16], by trying to answer the following question: *"What is the minimum knowledge that a process must have about the existence of others in order to solve the consensus problem in a system subject to up to f Byzantine failures?"* In answering this question, this paper presents the following contributions:

1) It redefines the k-OSR PD in order to establish even weaker conditions regarding the knowledge connectivity necessary for solving BFT-CUP;
2) It introduces the notion of *safe Byzantine failure pattern*, which refines previous results by considering the actual position of failed nodes in the knowledge connectivity graph, establishing thus the minimal conditions in which BFT-CUP is solvable;
3) It presents novel algorithms for showing that the safe Byzantine failure pattern is sufficient to solve the BFT-CUP problem.

Paper Organization. The remaining of the paper is organized in the following way. Section 2 presents some preliminary definitions. Section 3 describes a dissemination protocol. Section 4 describes the BFT-CUP protocol. Section 5 proves the necessary conditions to solve BFT-CUP. Finally, Section 6 presents final remarks.

2 Preliminaries

2.1 System Model

We consider a distributed system composed by a finite set Π of processes (also called participants or nodes) drawn from a larger universe $\mathcal{U}$. In a *known network*, Π is known to every participating process, while in an *unknown network*, a process $i \in \Pi$ may only be aware of a subset $\Pi_i \subseteq \Pi$.

Processes are subject to *Byzantine failures* [4]. A process that does not follow its algorithm in some way is said to be *faulty*. A process that is not faulty is said to be *correct*. Despite the fact that a process does not know all participants of the system (i.e., Π), it does know the expected maximum number of faulty process in Π, denoted by f. We define F as the set of processes in the system that actually have failed, F is unknown and $|F| \leq f$. We assume that all processes have a unique id, and that it is infeasible for a faulty process to obtain additional ids to launch a *sybil attack* [21].

Processes communicate by sending and receiving messages through *authenticated and reliable point to point channels*. Authenticity of messages disseminated to a not yet known process is verified through message channel redundancy, as explained in Section 3. A process i may only send a message directly to another process j if $j \in \Pi_i$, i.e., if i knows j. Of course, if i sends a message to j such that $i \notin \Pi_j$, upon receipt of the message, j may add i to Π_j, i.e., j now knows i and become able to send messages to it. We assume the existence of an underlying routing layer resilient to Byzantine failures [22], [23], in such a way that if $j \in \Pi_i$ and there is sufficient network connectivity, then i can send a message reliably to j.

Our protocol does not require any assumption about the relative speed of processes or message transfer delays (asynchronous systems). However, our protocol uses an underlying (standard) Byzantine consensus black box. Such primitive can be implemented in an eventually synchronous system (e.g., [8], [10]) or in a completely asynchronous system (e.g., using randomization [2], [5], [24], [25]). Consequently, our algorithms do not require any additional synchrony than what is required by the underlying consensus primitive.

2.2 Participant Detectors

To solve any nontrivial distributed coordination task, processes must somehow get a partial knowledge about the others. The *participant detector* oracle, namely PD, was proposed to handle this subset of known processes [12]. It can be seen as a distributed oracle that provides hints about the participating processes in the computation. Let $i.PD$ be defined as the participant detector of a process i. When queried by i, $i.PD$ returns a subset of processes in Π to which i can send messages.

Participant detectors provide an initial list of participants through which it is possible to expand the knowledge about Π. Notice that Byzantine processes can selectively hide the knowledge they possess or forge their knowledge about other participants. We say a participant p is a *neighbor* of another participant i if and only if $p \in i.PD$.

The information provided by the participant detectors of all processes form a *knowledge connectivity graph*, which is directed since the PD initial knowledge is not necessarily bidirectional [12].

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 $(i, j) \in E$ if and only if $j \in i.PD$, i.e., i knows j.*

It is important to remark that the knowledge connectivity graph defines the list of processes that a process initially knows in the system, *not the connectivity of the network*. As described in Section 2.1, we assume an underlying routing layer that allow processes to communicate.

Based on the properties of G_{di}, some classes of participant detectors have been proposed to solve CUP [12] and FT-CUP [13], [14]. The *k*-OSR (*k*-One Sink Reducibility) PD was proposed by [14] to solve FT-CUP with minimal synchrony assumptions and also has been used in a previous work to solve BFT-CUP [16]. In this paper, we redefine the k-OSR PD in order to establish even weaker knowledge connectivity conditions for solving BFT-CUP.

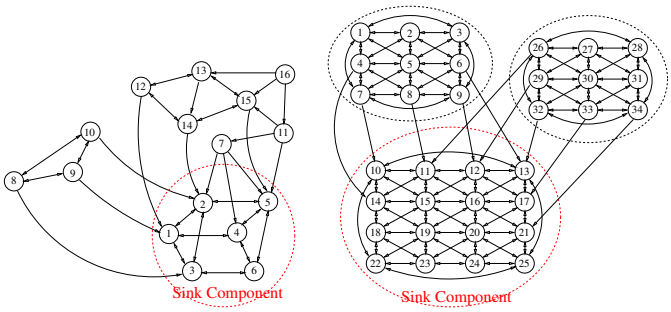
Before presenting the new *k*-OSR PD definition, we need to introduce some graph notations. Let $G = (V, E)$ be the *undirected graph* representing the knowledge relation determined by the PD oracle. Then, $V = \Pi$ and $(i, j) \in E$ if and only if $j \in i.PD$ or $i \in j.PD$, i.e., i knows j or j knows i. The *undirected graph* obtained from the directed knowledge connectivity graph $G_{di} = (V_{di}, E_{di})$ is defined as $G = (V_{di}, \{(i, j) : (i, j) \in E_{di} \vee (j, i) \in E_{di}\})$. We say that a subgraph G_c of G_{di} is *k-strongly connected* if for any pair (i,j) of nodes in G_c, i can reach j through at least *k* node-disjoint paths in G_c[1]. A component $G_{sink} = (V_{sink}, E_{sink})$ of G_{di} is a *sink component* if and only if there is no path from a node in G_{sink} to other nodes of G_{di}, except nodes in G_{sink} itself. Finally, a participant $p \in G_{di}$ is a *sink participant* if and only if $p \in G_{sink}$, otherwise p is a *non-sink participant*.

1. Recall that G_c is not a communication network graph but it represents the knowledge of processes; consequently, the notion of k-strongly connected means that there are enough knowledge connectivity in G_c for processes to reach each other, i.e., there are at least k node-disjoint paths.

Definition 2 *(**k-One Sink Reducibility PD (**k-OSR**)**). This class of **PD** contains all knowledge connectivity graphs G_{di} such that:*

1) *the undirected graph G obtained from G_{di} is connected;*
2) *the directed acyclic graph (DAG) obtained by reducing G_{di} to its strongly connected components has exactly one sink, namely G_{sink};*
3) *the sink component G_{sink} is k-strongly connected;*
4) *for all i, j, such that $i \notin G_{sink}$ and $j \in G_{sink}$, there are at least k node-disjoint paths from i to j.*

If G_{di} is a knowledge connectivity graph that satisfy the k-OSR PD definition, we say that $G_{di} \in k$-OSR. Figure 1 presents two knowledge connectivity graphs satisfying the k-OSR definition, for $k = 3$ and $k = 5$. For example, in Figure 1(a), the value returned by $1.PD$ is the subset $\{2, 3, 4\}$ of Π, meaning that process 1 initially knows processes 2, 3 and 4.



(a) 3-OSR PD (b) 5-OSR PD

Figure 1. Knowledge connectivity graphs satisfying k-OSR PD definition.

In our algorithms we assume that each process i queries its participant detector $i.PD$ exactly once at the beginning of the protocol execution. This means that the partial snapshot made by processes about the knowledge relationship is made once for all processes, so that there will be one graph G_{di} representing the system at the start of the protocols execution. Indeed, the union of the initial queries defines a single knowledge connectivity graph G_{di}. The main objective of this paper is to shed light in the minimal properties required from G_{di} for solving Byzantine consensus. Furthermore, processes do not know others initial views, which means that each one of them may obtain only a subgraph of G_{di}.

2.3 The Safe Byzantine Failure Pattern

Previous works showed that to solve both FT-CUP [14] and BFT-CUP [16], it is necessary that G_{di} satisfy the k-OSR PD condition. In these works, the connectivity parameter k is chosen in a conservative way, always considering the worst scenario for all participants in the system and all combinations of failures. However, this value can be relaxed in accordance with the position of faulty processes in G_{di}. More specifically, the previous proposed solution for BFT-CUP [16] does not consider the dynamism of the failures in the system, that is, it does not account for the actual pattern of failures in G_{di}, and defines bounds for the worst case scenario: the degree of connectivity as $k \geq 2f + 1$ in order

to tolerate up to f Byzantine failures. This means that each process must have at least $2f + 1$ node-disjoint paths to all other processes in G_{sink}. However, as we will show in this paper, if there are $f + 1$ node-disjoint paths *composed by correct processes* connecting these processes, then BFT-CUP admits solution. This means that during execution, depending on the location of the f failures in G_{di}, weaker conditions are necessary for solving BFT-CUP. In this sense, the minimum knowledge about the system composition can be expressed not only by taking into account the knowledge connectivity of processes, but also the actual location of failures in $G_{di} \in k$-OSR PD.

To better illustrate this idea, consider the 5-OSR graph presented in Figure 1(b). The previous solution for BFT-CUP [16] states that it is possible to tolerate up to two malicious failures in that scenario ($k \geq 2f+1, k = 5, f = 2$). However, a 3-OSR graph is sufficient to solve BFT-CUP in an execution in which nodes 4 and 10 are faulty, as illustrated in Figure 2.

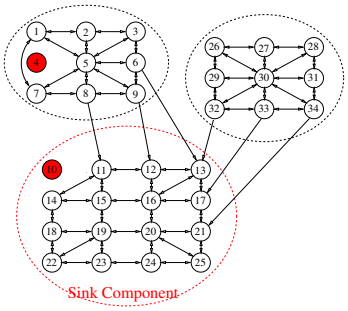


Figure 2. Safe Byzantine Failure Pattern ($f = 2$).

Notice that, the knowledge that processes have about the system is greater in the graph presented in Figure 1(b). Besides that, in the previous solution [16], non-sink participants must be grouped into k-strongly connected components, a condition that is not necessary in the redefined k-OSR (Definition 2). To represent this decrease in the required knowledge about the system composition, we define the notion of a *safe Byzantine failure pattern*.

Definition 3 ***(Safe Byzantine Failure Pattern)***. *Let G_{di} be a knowledge connectivity graph, f be the maximum number of processes in G_{di} that may fail and F be the set of faulty processes in G_{di} during an execution, we define the* safe Byzantine failure pattern *for G_{di} and F as the graph $G_{safe} = G_{di} \setminus F : (F \subset G_{di}) \wedge (|F| \leq f) \wedge (G_{di} \setminus F \in (f+1)\text{-}OSR)$.*

We say that a graph G_{di} is *Byzantine-safe* for F if its safe Byzantine failure pattern holds during the execution, i.e., if G_{safe} exists. Notice that this pattern ensures that whatever the actual location of the failures in G_{di} (i.e., the set of nodes in F), G_{safe} satisfies the $(f + 1)$-OSR PD properties. Consequently, G_{safe} contains at least $f + 1$ node-disjoint paths composed by *correct processes* between processes in G_{sink} and between a process outside G_{sink} to a process inside it.

Differently from the knowledge connectivity conditions stated in [16], the safe Byzantine failure pattern defines connectivity conditions that consider the actual location of the failures in the graph (although processes do not know these locations). In this way, it contains graphs that may not satisfy the conditions stated in [16], but do allow the BFT-CUP resolution. Consequently, the pattern refines the previous minimal knowledge conditions by considering all possible graphs in which the BFT-CUP can be solved, despite the occurrence of up to f faults.

2.4 The Consensus Problem

The consensus problem consists of ensuring that all correct processes of a distributed system eventually decide the same value, previously proposed by some process. Thus, each process i *proposes* a value v_i and all correct processes *decide* on some unique value v among the proposed values. Formally, consensus is defined by the following properties (e.g., [1]):

- *Validity*: if a correct process decides v, then v was proposed by some process.
- *Agreement*: no two correct processes decide differently.
- *Termination*: every correct process eventually decides some value.[2]
- *Integrity*: every correct process decides at most once.

The BFT-CUP problem corresponds to the consensus in unknown networks (CUP) with the additional requirement that a bounded number of participants can be subject to Byzantine failures.

3 Reachable Reliable Broadcast

This section introduces a new primitive, namely *reachable reliable broadcast*, used by processes to communicate. This primitive is generic enough to be used in any system where processes do not know all participants of the computation and need to broadcast messages reliably. In this paper, it will be used in the solution of BFT-CUP. Before defining how processes invoke the primitive, let us define the notion of *f-reachability*.

Definition 4 ***(f-reachability)***. *Consider G_{di} a knowledge connectivity graph and let f be the number of nodes in G_{di} that may fail. For any two participants $p, q \in G_{di}$, q is* f-reachable *from p in G_{di} if there are at least $f + 1$ node-disjoint paths from p to q in G_{di} composed only by correct processes.*

Let m be a message, processes access the reachable reliable broadcast primitive by invoking two basic operations:

- reachable_bcast(m,p) – through which the participant p broadcasts m to all f-reachable participants from p in G_{di}.
- reachable_deliver(m,p) – invoked by a receiver to deliver m sent by the participant p.

The reachable reliable broadcast should satisfy the following properties:

- *RB_Validity*: If a correct participant p invokes reachable_bcast(m,p) then (i) some correct participant q,

2. In case a randomized protocol is used as underlying Byzantine consensus, the termination is ensured with probability 1 [2], [5], [24].

f-reachable from p in G_{di}, eventually invokes reachable_deliver(m,p) (ii) or there is no correct participant f-reachable from p in G_{di}.
- *RB_Integrity*: For any message m, if a correct participant q invokes reachable_deliver(m,p) then some participant p has invoked reachable_bcast(m,p).
- *RB_Agreement*: If a correct participant q invokes reachable_deliver(m,p), where m was sent by a correct process p that invoked reachable_bcast(m,p), then all correct participants f-reachable from p in G_{di} invoke reachable_deliver(m,p).

These properties establish a communication primitive with specification similar to the usual reliable broadcast [1], [2], [24]. Nonetheless, this primitive only ensures the delivery of messages to the correct processes that are f-reachable from a correct sender in G_{di}. More specifically, our agreement property differs from standard agreement property due to the lack of knowledge that processes have about the system. In this model, a malicious process is able to send a message m using only some paths in G_{di} in a way that only a subset (and not all) of the correct processes reachable from it will deliver m.

In this sense, our primitive is weaker than classical reliable or echo broadcast primitives, but it is enough to solve BFT-CUP: the DISCOVERY sub-protocol does not require agreement in the messages sent by malicious processes (see Section 4).

3.1 The Reachable Reliable Broadcast Protocol

Algorithm 1 presents an implementation for the reachable reliable broadcast primitive. Its main idea is that participants flood their messages to all f-reachable processes, which, in turn, deliver these messages as soon as their authenticity has been proved.

Notations. The algorithm uses the following notations:

- $i.received\_msgs$ – set containing tuples of the form $\langle m, m.route\rangle$ in which m is a message received by process i and $m.route$ is an ordered list of processes that have received m. The first element of $m.route$ contains the id of its original sender.
- $computeDisjointRoutes(m, i.received\_msgs)$ – a function that receives as input a message m and a set of routes from where m was received at participant i and computes the number of node-disjoint paths through which m has been received at i.
- $appendRoute(m.route, i)$ – a function that adds i to the end of $m.route$;
- $getFirstElement(m.route)$, $getLastElement(m.route)$ – functions that return the first and last process id of $m.route$, respectively.

Description. A process i broadcasts a message m by the invocation of reachable_bcast(m, i) (line 6). In this case, through a RC_FLOODING$(m, m.route = [i])$ message, i sends m to its neighbors, i.e., the processes returned by its participant detector. The message carries the $m.route$ list, that is initialized with i and contains the accumulated route according with the path traversed from the sender to a receiver.

When RC_FLOODING$(m, m.route)$ is received by i (from j), the content of the message is first evaluated in lines 8-16. If its content is valid, process i forwards m to its neighbors, except j. This implements the flooding of m in such a way that it will arrive at all f-reachable participants from the sender (line 17).

During the evaluation of the contents of RC_FLOODING$(m, m.route)$, i initially certifies that m has been actually sent by j and that it has not yet been received by itself (line 8). Then, i appends its id to $m.route$ (line 9) and stores m together with the $m.route$ in the $i.received\_msgs$ bag (line 10). Finally, i delivers m if and only if it has received m through $f + 1$ node-disjoint paths, i.e., the authenticity of m has been verified since it was received by at least one path composed only by correct processes. This is done using the $computeDisjointRoutes$ function (line 11). If that is the case, i calls reachable_deliver$(m, initiator)$ to deliver m sent by the $initiator$ and then removes it from its $i.received\_msgs$ bag (line 15).

Algorithm 1 Reachable Reliable Broadcast (participant i).

```
constant:
 1) f : int                                   // upper bound on the number of failures

variables:
 2) i.received_msgs : bag of ⟨m, m.route⟩ tuples

message:
 3) RC_FLOODING:                              // struct of this message
 4)     m : message to flood                  // value to be disseminated
 5)     route : ordered list of nodes         // path traversed by m

** Initiator Only **
upon invocation of reachable_bcast(m, i)
 6) ∀j ∈ i.PD, send RC_FLOODING(m, m.route = [i]) to j;

** All Nodes **
INIT:
 7) i.received_msgs ← ∅;

upon receipt of RC_FLOODING(m, m.route) from j
 8) if getLastElement(m.route) = j ∧ i ∉ m.route then
 9)     appendRoute(m.route, i);
10)     i.received_msgs ← i.received_msgs ∪ {⟨m, m.route⟩};
11)     routes ← computeDisjointRoutes(m, i.received_msgs);
12)     if routes ≥ f + 1 then
13)         initiator ← getFirstElement(m.route);
14)         trigger reachable_deliver(m, initiator);
15)         i.received_msgs ← i.received_msgs \ {⟨m, ∗⟩};
16)     end if
17)     ∀z ∈ i.PD \ {j}, send RC_FLOODING(m, m.route) to z;
18) end if
```

The solution presented herein is based on the approach of [26] and it enforces that each participant appends itself at the end of the routing information in order to send or forward a message. A participant will process a received message only if the participant that is sending (or forwarding) this message appears at the end of the accumulated route. Nonetheless, a malicious participant is able to modify the accumulated route (removing or adding participants) and to modify or block the message being propagated. However, the connectivity degree ensures that messages will be received at all f-reachable participants (there will be at least $f + 1$ node-disjoint paths composed only by correct processes).

Our primitive needs only $f + 1$ correct node-disjoint paths ($2f+1$ if we consider that f paths contain some faulty process) because *RB_Agreement* considers only messages broadcast by correct processes. Consequently, a message

sent by a malicious process may be delivered only by some processes, but not all. In a standard Byzantine reliable broadcast algorithm [24], which requires at least $3f + 1$ processes, a message broadcast by a faulty process is either (1) delivered by all correct processes or (2) not delivered by any processes.

3.2 Reachable Reliable Broadcast Correctness

Algorithm 1 has the drawback that a message may be delivered more than once by its receivers, but this does not affect its correctness. Moreover, its properties are sufficient to solve BFT-CUP (Section 4). Let us prove the correctness of the *reachable reliable broadcast* algorithm.

Lemma 1 *(**RB_Validity**) If a correct participant p invokes reachable_bcast(m,p) then (i) some correct participant q, f-reachable from p in G_{di}, eventually invokes reachable_deliver(m,p) or (ii) there is no correct participant f-reachable from p in G_{di}.*

Proof. Let us first prove Case (i). From Definition 4, since q is f-reachable from p, there are at least $f + 1$ node-disjoint paths in G_{di} composed only by correct nodes from p to q. Let $P : p = 0, 1, ..., k = q$ be one of those paths. Let us prove by induction on k that q will receive the message m sent through P. The base case ($k = 1$) is trivial, since p is correct and then it will invokes reachable_bcast(m,p) to all its neighbors returned from $p.PD$ (line 6). By the induction step, the claim is valid for process i in P ($k = i$). Then, on the reception of m by i, predicate of line 8 will be satisfied since moreover processes in the path are correct. Then, i will execute line 17 sending m to all its neighbors including $i+1 \in i.PD$ ($k = i+1$); since channels are reliable, $i+1$ will receive m and the claim follows. Since there are at least $f+1$ node-disjoint paths in G_{di} composed only by correct nodes from p to q, it is ensured that q receives m through the $f+1$ node-disjoint paths (including P), thus satisfying predicate of line 8 at least $f + 1$ times. Then, the authenticity of m can be verified at q through redundancy. This is done by the execution of lines 9–11, which are responsible to maintain information regarding the different routes from which m has been received at q. Whenever the message authenticity is proved (line 12) the delivery of m is authorized at q by the invocation of reachable_deliver(m,p) (line 14). This proves that if q is a correct node f-reachable from p in G_{di}, then q delivers m at least once.

Case *(ii)* can be proved by exactly the same arguments of Case *(i)*: since all correct participant disseminate m to all its neighbors (line 17), if m is not delivered by some correct participant in G_{di}, then there is no correct participant which is f-reachable from p in G_{di}. □

Lemma 2 *(**RB_Integrity**) For any message m, if a correct participant q invokes reachable_deliver(m,p) then some participant p has invoked reachable_bcast(m,p).*

Proof. Consider that a correct participant q invokes reachable_deliver(m,p), then q received m through $f + 1$ node-disjoint paths from p (lines 8-16). Now, we have to prove that p has invoked reachable_bcast(m,p). Assume, for the sake of contradiction, that p has not invoked reachable_bcast(m,p). In this case, in order to receive m at q through some path $P : p, ..., z, ..., q$, a malicious participant z needs to forge the dissemination of m from p. As there are at most f malicious participants, m will be received at q from at most f node-disjoint paths (each of these f paths may contain one malicious participant) and q never will invoke reachable_deliver (m,p), reaching a contradiction. Consequently, if a correct participant q invokes reachable_deliver(m,p), then some participant p has invoked reachable_bcast(m,p). □

Lemma 3 *(**RB_Agreement**) If a correct participant q invokes reachable_deliver(m,p), where m was sent by a correct process p that invoked reachable_bcast(m,p), then all correct participants f-reachable from p in G_{di} invoke reachable_deliver(m,p).*

Proof. From Lemma 2, if a correct participant q invokes reachable_deliver(m,p), then some participant p has invoked reachable_bcast(m,p). From Lemma 1, if a correct participant p invokes reachable_bcast(m,p), then some correct participant q, f-reachable from p in G_{di}, eventually invokes reachable_deliver(m,p). By generalization, all correct participants f-reachable from p in G_{di} invoke reachable_deliver(m,p). □

4 BFT-CUP: Byzantine Consensus with Unknown Participants

This section presents an algorithm for solving BFT-CUP under the safe Byzantine failure pattern assumption. As previously stated, we assume that there is an underlying routing layer able to deliver messages reliably between known processes despite Byzantine faults and asynchrony. Besides this communication infrastructure, our solution uses the *reachable reliable broadcast* primitive described in previous section and a *standard Byzantine consensus* black box (e.g., [10]).

Using these building blocks and the participant detector abstraction for getting some initial knowledge about the participants of the system, the BFT-CUP protocol is divided in three sub-protocols (see Figure 3). The DISCOVERY sub-protocol (Section 4.1) is used by each participant to increase its knowledge about other processes in the system. In the SINK sub-protocol, each participant discovers if it belongs to the sink component or not (Section 4.2). In the last sub-protocol, CONSENSUS, the participants in the sink execute a standard Byzantine consensus and disseminate the decision value to non-sink participants (Section 4.3).

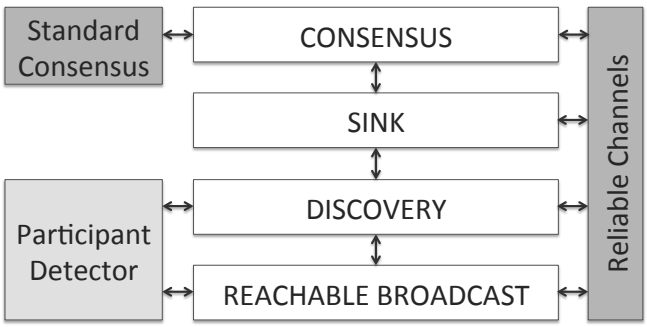


Figure 3. Building blocks and sub-protocols of the BFT-CUP algorithm.

The protocols discussed in this section consider the following assumptions beside the ones described in our system model.

Assumption 1 *(Knowledge connectivity) The knowledge connectivity graph G_{di} is* Byzantine-safe *for F.*

Assumption 2 *(BFT consensus) The necessary conditions to execute a standard BFT consensus among processes in G_{sink}, namely, it contains at least $2f + 1$ correct processes.*

4.1 Participants Discovery

The first step to solve consensus in a system with unknown participants is to provide processes with the maximum possible knowledge about the system composition.

4.1.1 The DISCOVERY Protocol

The main idea behind the algorithm (Algorithm 2) is that each participant i runs a kind of breadth-first search in G_{di}, where i broadcasts a message requesting information about neighbors of each participant f-reachable in G_{di}. An important characteristic of this algorithm is that it is only ensured to terminate at sink participants, i.e., non-sink participants may not terminate the execution of the protocol. In these cases, a non-sink participant will still be able to discover all sink participants, which is enough to obtain the value decided in the sink and terminate. A participant that terminates this algorithm will obtain a partial view of the system, composed by the maximal set of participants f-reachable from it in G_{di}. In this way, this algorithm ensures the following properties: *(1)* each sink participant i terminates the protocol by discovering exactly G_{sink}, i.e., it returns $i.known = V_{sink}$; *(2)* each non-sink participant i discovers G_{sink}, i.e, eventually $i.known \supset V_{sink}$; and *(3)* each non-sink participant i that terminates this protocol obtains strictly more knowledge than a sink participant, i.e., it returns $i.known \supset V_{sink}$.

Notations. The algorithm uses the following notations:

1) $i.known$ – set containing ids of all processes known by i.
2) $i.received$ – set containing ids of processes that sent a reply message (SET_NEIGHBOR) to i.
3) $i.msg\_pend$ – set containing ids of processes that should send a message to i, i.e., for each $j \in i.msg\_pend$, i should receive a message from j.
4) $i.nei\_pend$ – set of tuples $\langle j, j.neighbor\rangle$, where $j.neighbor$ contains ids of possible neighbors of j. It represents a process j that i knows but it did not got enough information to be sure that all processes in $j.neighbor$ really exists.
5) $\#_{\langle *,j\rangle} i.nei\_pend$ – number of tuples $\langle *, *.neighbor\rangle \in i.nei\_pend$ with $j \in *.neighbor$, i.e., number of different processes that reported to i that j is in their neighborhood.
6) $i.asked$ – set containing the ids of processes that asked i about the decision.
7) $i.decision$ – variable containing the consensus decision value (set up during the execution of Algorithm 4).

Algorithm 2 DISCOVERY code of participant i.

```
constant:
1)  f : int                                   // upper bound on the number of failures

variables:
2)  i.known : set of nodes                                   // known nodes
3)  i.nei_pend : set of ⟨node, node.neighbor⟩ tuples         // i does not
    know all neighbors of node
4)  i.msg_pend : set of nodes        // nodes that i is waiting for replies
5)  i.received : set of nodes         // nodes that i has received a reply
6)  i.asked : set of nodes        // nodes that have required the decision
    value
7)  i.decision : value                                       // decision value

message:
8)  GET_NEIGHBOR
9)  SET_NEIGHBOR:
        neighbor : set of nodes          // neighbors of the sending node
10) SET_DECISION:
        decision : value                                     // the decided value

Task MAIN
11) i.known, i.msg_pend ← {i} ∪ i.PD;
12) i.nei_pend, i.received, i.asked ← ∅;
13) Fork DELIVER();
14) reachable_bcast(GET_NEIGHBOR, i);

Task DELIVER
15) upon execution of reachable_deliver(GET_NEIGHBOR, p);
16) if i.decision =⊥ then // i has not decided yet
17)     i.asked ← i.asked ∪ {p};
18) else // i already decided
19)     send SET_DECISION(i.decision) to p;
20) end if
21) send SET_NEIGHBOR(i.PD) to p;

22) upon receipt of SET_NEIGHBOR(p.neighbor) from p
23) i.received ← i.received ∪ {p};
24) i.msg_pend ← i.msg_pend \ {p};
25) i.nei_pend ← i.nei_pend ∪ {⟨p, p.neighbor⟩};
26) for all j: (#⟨*,j⟩ i.nei_pend > f) ∧ (j ∉ i.known) do
27)     i.known ← i.known ∪ {j};
28)     if {j} ∉ i.received then
29)         i.msg_pend ← i.msg_pend ∪ {j};
30)     end if
31) end for
32) for all ⟨j, j.neighbor⟩ ∈ i.nei_pend do
33)     if (∀z ∈ j.neighbor: z ∈ i.known) then
34)         i.nei_pend ← i.nei_pend \ {⟨j, j.neighbor⟩};
35)     end if
36) end for
37) if (|i.nei_pend| + |i.msg_pend|) ≤ f then
38)     return i.known;
39) end if
```

Description. In the initialization of Algorithm 2, the sets $i.known$ and $i.msg\_pend$ are updated according with the neighbors returned by the participant detector $i.PD$ (line 11). Then, i broadcasts (using Algorithm 1) a message GET_NEIGHBOR, requesting information about system composition to all participants f-reachable from it in G_{di} (line 14). The *Task* DELIVER is launched in line 13 to deal with the delivery of such a message and to disseminate the decision of consensus if it has already been taken.

When i delivers a message GET_NEIGHBOR sent by a participant p (line 15), it sends back to p a reply SET_NEIGHBOR, indicating its neighbors (line 21). Moreover, this message carries also a request for the decision value: if i already knows the decision value, it sends this value to p; otherwise i stores the identifier of p in $i.asked$ in order to be able to send the decision to it as soon as it gets to know the decided value (Algorithm 4).

Upon receipt of a SET_NEIGHBOR message from p (line 22), i updates the sets of received replies, pending neighbors and pending messages with p (lines 23–25) and verifies whether i has acquired knowledge about any new participant (lines 26–31). To ensure safety, i gets to know a

participant j if and only if at least $f+1$ other processes reported to i that j is their neighbor (line 26). After this verification, the set of pending neighbors is updated (lines 32–36), according to the new participants discovered.

In order to decide if there is still some participant to be discovered, i uses the $i.nei\_pend$ and $i.msg\_pend$ sets, which store the pending messages related to replies received by i. The algorithm ends when there remains at most f pending messages (lines 37–39). The intuition behind this condition is that by assuming the safe Byzantine failure pattern there will be enough knowledge connectivity to ensure that if there are at most f pending messages at process i, then i has already discovered all processes f-reachable from it in $G_{di} \in k$-OSR (see Lemmata 6, 7 and 8). Consequently, the algorithm ends by returning the set of participants discovered by i, which contains all participants (correct or faulty) f-reachable from i in G_{di}.

Termination at non-sink participants. As mentioned before, Algorithm 2 may not terminate in a participant that is not in the sink G_{sink} of G_{di}. Consider two participants p, q such that $p, q \in G_{di} \setminus G_{sink}$ and q is f-reachable from p (Definition 4). The fact that q is f-reachable by p does not imply that all the neighbors of q are f-reachable by p. It may happen that some neighbors of q could not deliver the GET_NEIGHBOR message sent by p and thus will not send a reply to p, remaining in the $p.nei\_pend$ set. Consequently, the number of pending replies at p could never be lower (or equal) to the threshold f (line 37). Hopefully, if that is the case, p can still wait for the decision value that will be sent to it by the processes that are f-reachable from it in G_{di} (at least all processes in G_{sink}).

Figure 4 presents a scenario for a 2-OSR PD ($f = 1$, no failures), where Algorithm 2 does not terminate at non-sink participant 1. This happens because, although participants 2 and 3 are f-reachable from 1 (actually $2, 3 \in 1.PD$), participants 4 (neighbor of 2) and 5 (neighbor of 3) are not f-reachable from 1 and will never deliver the GET_NEIGHBOR message from 1. Consequently, 2 and 3 remain in $1.nei\_pend$ forever and, as $f = 1$, the algorithm does not terminate at non-sink participant 1. Fortunately, this does not happen with sink participants, since all participants in G_{sink} are f-reachable from any participant in G_{di} and by the k-OSR PD properties (Definition 2), processes in G_{sink} only have neighbors that also belong to G_{sink}.

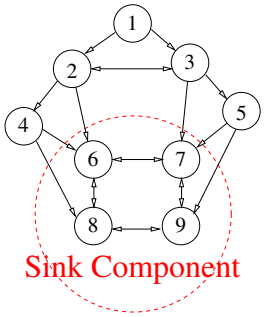


Figure 4. G_{di} generated by a 2-OSR PD (no failures).

4.1.2 DISCOVERY Correctness

The DISCOVERY protocol uses the *reachable reliable broadcast* primitive to discover the participants in the sink G_{sink} of G_{di}. We start by proving two lemmata that shows this is indeed true if the safe Byzantine failure pattern is assumed.

Lemma 4 *(**Sink Participants Reachability**) Under Assumption 1, each node $q \in G_{sink}$ is f-reachable from any node $p \in G_{di}$.*

Proof. From the definition of safe Byzantine failure pattern (Definition 3), there are at least $k \geq f+1$ node-disjoint paths composed by correct processes from any node $p \in G_{di}$ to each node $q \in G_{sink}$. Then, by f-reachability definition (Definition 4), the lemma follows. □

Lemma 5 *(**Sink Participants Messages Delivery**) Under Assumption 1, the* REACHABLE RELIABLE BROADCAST *primitive ensures that messages from any correct process $p \in G_{di}$ will be delivered to every correct process $q \in G_{sink}$.*

Proof. From Lemma 4, each participant $q \in G_{sink}$ is f-reachable from every participant $p \in G_{di}$. Thus, this proof follows directly from Lemmata 1 (*RB_Validity*), 2 (*RB_Integrity*) and 3 (*RB_Agreement*). □

Algorithm DISCOVERY satisfies some properties stated by Lemmata 6, 7 and 8. Before proceeding with the proofs, let us make two important observations about the algorithm.

Observation 1 *From Lemma 4 and the properties of $G_{di} \setminus F \in k$-OSR, we have that (i) every node $z \in G_{sink}$ is f-reachable from every $p \in G_{di}$ (Definition 4); (ii) if $p \in G_{sink}$ then only nodes in G_{sink} are f-reachable from p; and (iii) every $z \in G_{sink}$ (correct or faulty) is known by at least $f+1$ correct neighbors, thus, $z \in PD$ of at least $f+1$ correct nodes.*

Observation 2 *For a process j, $j \in i.known$ if: (1) $j \in i.PD$ (from line 11); or (2) Let $X := \{q | j \in q.PD \wedge q$ is f-reachable from $i\}$, then $|X| > f$ (from lines 11, 14–15, 21–22, 26–27), i.e., there are more than f processes f-reachable from i that know j and reported this to i, satisfying the predicate of line 26.*

Lemma 6 *Under Assumption 1, a correct participant $i \in G_{di}$ executing algorithm DISCOVERY eventually discovers all participants in G_{sink}.*

Proof. From Lemma 5, when a correct node i executes a reachable_bcast(GET_NEIGHBOR, i) (line 14), every correct node $q \in G_{sink}$, calls reachable_deliver(GET_NEIGHBOR, i) (line 15) and sends a SET_NEIGHBOR message containing $q.PD$ in response (line 21). Due to the assumption of reliable channels between every pair of processes, this message will be received by i (line 22). Now, let us prove that i keeps receiving these messages until it collects enough information to discover all participants in G_{sink}.

For every node $q \in G_{sink}$, there are at least $f+1$ correct node-disjoint paths from i to q (Assumption 1). Since all neighbors of i (including the $f+1$ present in such paths) are in $i.msg\_pend$ at the beginning of execution (line 11), the algorithm only ends at i after at least one of these participants

has been removed from $i.msg\_pend$. This happens because the algorithm ends when $|i.nei\_pend| + |i.msg\_pend| \leq f$ (line 37). Let P be one of the correct node-disjoint paths from i to q, $P : i = 0, z = 1, x = 2, ..., n = k, q$; we will prove by induction on k that q must be discovered by i before termination. For the base case ($k = 1$), node $z \in i.PD$ is removed from $i.msg\_pend$ when i received the reply SET_NEIGHBOR from z (line 24), but when this happens $z \in i.nei\_pend$ (line 25) and the algorithm does not finish because z is still in a pending set of i. Since node $x \in z.PD$, z is only removed from $i.nei\_pend$ after i had discovered x (lines 33-35). When this happens, $x \in i.msg\_pend$ (lines 26-31) and the algorithm does not finish because now x is in a pending set of i. By the induction step, the claim is valid for node n in P ($k = n$). Then, $n \in (k-1).PD$ is removed from $i.msg\_pend$ when i received the reply SET_NEIGHBOR from n, but when this happens $n \in i.nei\_pend$ and the algorithm does not finish because now n is in a pending set of i. Since $q \in n.PD$, n is only removed from $i.nei\_pend$ after i had discovered q (lines 33-35) and the claim follows. By generalization, the algorithm does not finish at a correct node i before it had discovered all participants in G_{sink}.

Since we proved that node i does not finish the algorithm before it had discovered all participants in G_{sink}, we can consider that i eventually reach a state in which: for every correct node $q \in G_{sink}$, $q \in i.received \wedge q \notin i.msg\_pend$ (from lines 11–12, 23–24, 27–29) and; for a malicious or a crashed node $x \in G_{sink}$ that does not sent back a reply SET_NEIGHBOR to i, $x \notin i.received \wedge x \in i.msg\_pend$ (from Observations 1 and 2 and lines 11–12, 23–24, 27–29). In both situations, i will receive SET_NEIGHBOR ($neigh$) messages from at least $f + 1$ correct neighbors of x, q in which x,$q \in neigh$. Then, the predicate of line 26 is satisfied, and thus, every $x, q \in i.known$ (line 27). By generalization, i eventually discovers all participants in G_{sink}. □

Lemma 7 *(**Sink Participants – DISCOVERY**) Under Assumption 1, algorithm DISCOVERY executed by a correct node $i \in G_{sink}$ satisfies the following properties:*

- *Sink Termination: i terminates the execution;*
- *Sink Accuracy: i returns a set $i.known = V_{sink}$.*

Proof. We start by proving Sink Accuracy before proceed to Sink Termination.

Sink Accuracy: From Lemma 6, for every $p \in G_{sink}$ we have that $p \in i.known$. Now, let us prove that a node $z \notin G_{sink}$ (not f-reachable from i) will not be in $i.known$. Suppose that a malicious node x gets to known i and sent a SET_NEIGHBOR ($x.neighbor$) message to i indicating its presence in the system and/or the presence of z ($z \in x.neighbor$). In this case, $x \in i.received$, $\langle x, x.neighbor \rangle \in i.nei\_pend$ (lines 23–25), but $x, z \notin i.known$ and $x, z \notin i.msg\_pend$, since at most f processes could report to i the knowledge about x, z and from Observation 2, the predicate of line 26 will not be satisfied.

Consequently, at the end of the algorithm, following Lemma 6 and Observations 1 and 2, we can conclude that (i) $i.known = V_{sink}$ (satisfying *Accuracy*); (ii) $\{V_{sink} \setminus F\} \supseteq i.received \subseteq \{V_{sink} \cup F\}$; (iii) $i.msg\_pend \subseteq F$, $i.msg\_pend = \{i.known \setminus i.received\}$, $|\{i.known \setminus i.received\}| \leq f$, $|i.msg\_pend| \leq f$; and (iv) $i.nei\_pend \subseteq F$, $|i.nei\_pend| \leq f$.

Sink Termination: Now, let us prove that eventually $|i.nei\_pend| + |i.msg\_pend| \leq f$ and the algorithm terminates (line 37). From Assumption 1 and Observation 1, only nodes in G_{sink} are f-reachable from i and a node in G_{sink} is f-reachable from any other node in G_{sink}. Consequently, eventually $\forall j$ correct: $\langle j, * \rangle \notin i.nei\_pend$ and $j \notin i.msg\_pend$. Thus, $i.nei\_pend \cup i.msg\_pend \subseteq F$. Moreover, if $\langle j, * \rangle \in i.nei\_pend$ then $j \notin i.msg\_pend$ (from lines 11–12, 23–24). Thus, $|i.nei\_pend| + |i.msg\_pend| \leq f$, satisfying *Termination*. This concludes our proof and the lemma follows. □

Lemma 8 *(**Non-sink Participants – DISCOVERY**) Under Assumption 1, algorithm DISCOVERY executed by a correct node $i \notin G_{sink}$ satisfies the following properties:*

- *Non-sink Accuracy: Eventually $V_{sink} \subset i.known$;*
- *Non-sink Conditional Termination: If i terminates the algorithm, i returns $V_{sink} \subset i.known$.*

Proof. We start by proving Non-sink Accuracy before proceed to Non-sink Termination.

Non-sink Accuracy: At the beginning of execution $i.known = \{i\} \cup i.PD$ (line 11) and from Lemma 6, for every $p \in G_{sink}$ we have that $p \in i.known$. Consequently, eventually $i.known \supseteq \{i\} \cup i.PD \cup V_{sink}$ ensuring *Non-sink Accuracy*.

Non-sink Conditional Termination: Consider that the algorithn ends by returning $i.known$ (line 38). As eventually $i.known \supseteq \{i\} \cup i.PD \cup V_{sink}$, $V_{sink} \subset i.known$ and the lemma follows. □

4.2 Defining the Sink Component

The DISCOVERY sub-protocol eventually terminates in each sink participant, allowing them to discover all participants in G_{sink}. For non-sink participants, that protocol may terminate or not. Due to this, the second phase of our BFT-CUP protocol is necessary to determine which participants, from those who had finished the previous DISCOVERY, belong to G_{sink}. Recall that a process that does not terminate the previous phase does not belong to G_{sink}.

4.2.1 The SINK Protocol

This intermediary phase is represented by Algorithm 3 (SINK). It is executed by some process to determine whether it is a member of G_{sink} or not. It exploits the fact that after completing the DISCOVERY algorithm, the members of G_{sink} have the same partial view of the system (which is G_{sink}), whereas other participants have strictly more knowledge than these participants, i.e., each non-sink participant knows at least itself and the members of G_{sink}. In this way, this algorithm ensures the following properties: *(1)* each sink participant i terminates the protocol by returning $\langle true, V_{sink} \rangle$; and *(2)* if a non-sink participant i terminates DISCOVERY, then it also terminates SINK by returning $\langle false, i.known \rangle$, such that $i.known \supset V_{sink}$.

Notations. The algorithm uses the following notations:

1) $i.known$ – set containing ids of all processes known by i;

2) $i.nacked$ – set containing ids of nodes which are not in the same graph component of i in G_{di};
3) $i.acked$ – set containing ids of nodes which are in the same graph component of i in G_{di};
4) $i.in\_the\_sink$ – a boolean variable determining whether i is in the sink component.

Description. In the initialization phase (MAIN task), process i executes DISCOVERY in order to obtain its partial view of the system (line 10). Non-sink participants may never terminate this procedure, while sink participants finish it by discovering exactly all processes in G_{sink}. If i finishes DISCOVERY, it disseminates its set of known processes (line 14) to determine if it belongs to G_{sink} or not. When this message is delivered by process j, it replies with an *ack* to i if it has the same knowledge of i (i.e., j belongs to the same component of i). Otherwise, j replies a *nack* (lines 16-21). It is important to notice that j only replies to i after finishing DISCOVERY. This means that at least the correct processes in G_{sink} will reply.

Algorithm 3 SINK code of participant i

```
constant:
1) f : int                                   // upper bound on the number of failures

variables:
2) i.known : set of nodes                    // known nodes
3) i.nacked : set of nodes                   // not in the same component of i
4) i.acked : set of nodes                    // in the same component of i
5) i.in_the_sink : boolean                   // is i in the sink?

message:
6) REQUEST:
7)     known : set of nodes
8) RESPONSE:
9)     ack/nack : boolean

** All Nodes **
Task MAIN
10) i.known ← DISCOVERY();
11) i.acked ← {i};
12) i.nacked, ← ∅;
13) for all j ∈ i.known: j ≠ i do
14)     send REQUEST(i.known) to j;
15) end for

16) upon receipt of REQUEST(p.known) from p
17) if i.known = p.known then
18)     send RESPONSE(ack) to p;
19) else
20)     send RESPONSE(nack) to p;
21) end if

22) upon receipt of RESPONSE(m) from p
23) if m = nack then
24)     i.nacked ← i.nacked ∪ {p};
25)     if |i.nacked| > f then
26)         i.in_the_sink ← false;
27)         return ⟨i.in_the_sink, i.known⟩;
28)     end if
29) else
30)     i.acked ← i.acked ∪ {p};
31)     if |i.acked| ≥ |i.known| − f then
32)         i.in_the_sink ← true;
33)         return ⟨i.in_the_sink, i.known⟩;
34)     end if
35) end if
```

Upon receipt of a reply from a process p (line 22), two situations are possible: (1) if the reply is a *nack*, i adds p to the set $i.nacked$ of nodes belonging to other components; if the number of nodes in $i.nacked$ exceeds f, i concludes that it does not belong to G_{sink} and returns $false$ (lines 23–29). This condition holds because all nodes outside G_{sink} know all nodes in G_{sink}. Otherwise, (2) if the reply is an *ack*, i adds p to the set $i.acked$. Then, if i has received acks from all known processes, excluding the possible f faulty ones (line 31), it concludes that it belongs to G_{sink}. This condition holds because every process in G_{sink} receives replies only from members of G_{sink}. Moreover, in both cases, a collusion of f malicious participants cannot lead a process to decide incorrectly.

4.2.2 SINK Correctness

Lemmata 9 and 10 state properties satisfied by the SINK algorithm.

Lemma 9 *(**Sink Participants – SINK**) Under Assumption 1, algorithm SINK executed by a correct node $i \in G_{sink}$ satisfies the following properties:*

- *Sink Termination: i terminates the execution;*
- *Sink Accuracy: i returns $\langle true, V_{sink}\rangle$.*

Proof. By Lemma 7, every correct process $j \in G_{sink}$ terminates DISCOVERY; moreover, $\forall j, j.known = V_{sink}$. Then, i will receive RESPONSE (ack) messages (line 22) to its REQUEST (line 14) from every correct $j \in G_{sink}$, since moreover channels are reliable. On the occurrence of a collusion of f malicious processes that replies *nack* to i, at most $|i.nacked| \leq f$. Thus, predicate of line 25 will never be satisfied. Since the number of corrects is $|i.known| - f$, eventually predicate of line 31 is satisfied and i returns $\langle true, V_{sink}\rangle$, thus satisfying *Termination* and *Accuracy*. □

Lemma 10 *(**Non-sink Participants – SINK**) Under Assumption 1, algorithm SINK executed by a correct node $i \notin G_{sink}$ satisfies the following properties:*

- *Non-sink Conditional Termination: if i terminates DISCOVERY, then i terminates SINK as well;*
- *Non-sink Accuracy: if i terminates, it returns $\langle false, i.known\rangle$.*

Proof. If i terminates the execution of DISCOVERY, it sends a REQUEST to all nodes in $i.known$. By Lemma 8, $i.known \supset V_{sink}$. Thus, every correct process $j \in G_{sink}$ will receive the REQUEST from i and reply with a RESPONSE ($nack$), since $(j.known = V_{sink}) \neq i.known$, by Lemma 7, since moreover channels are reliable. From the properties of $G_{di} \setminus F \in k$-OSR and Lemma 4, there are at least $f + 1$ correct nodes in G_{sink}. Thus, i will receive in line 22 at least $f + 1$ responses carrying out $nack$ and the predicate of line 25 ($|i.nacked| \geq f + 1$) will be eventually satisfied. Moreover, i will never receive a number of replies with a RESPONSE (ack), such that $|i.acked| \geq |i.known| - f$, even on the occurrence of a collusion of f malicious processes that replies *ack* to i, and then the predicate of line 31 will never be satisfied. Thus, eventually i returns $\langle false, i.known\rangle$, satisfying *Conditional Termination* and *Accuracy*. □

4.3 Achieving Consensus

After processes discover whether they belong to G_{sink} or not, the processes in the sink execute a standard Byzantine consensus and then, afterwards, send the decision value to non-sink processes.

4.3.1 The CONSENSUS Protocol

Algorithm 4 presents the CONSENSUS protocol.

Notations. The algorithm uses the following notations:

- $i.known$ – set containing ids of all processes known by i;
- $i.in\_the\_sink$ – a boolean variable indicating whether i is in the sink component;
- $i.asked$ – set containing the ids of processes that asked i about the decision value. This value has been set up during the execution of Algorithm 2 (DISCOVERY);
- $i.decision$ – variable containing the decision value;
- $i.values$ – set of tuples of the type $\langle nodeid, value\rangle$;
- $\#_{\langle *,v\rangle}$ – the number of times that the decision value equals v appears in any tuple $\langle *, v\rangle \in i.values$.

Algorithm 4 CONSENSUS code of participant i

```
constant:
 1) f : int                                   // upper bound on the number of failures

input:
 2) i.initial : value                         // proposal value (input)

variables:
 3) i.in_the_sink : boolean                   // is i in the sink?
 4) i.known : set of nodes                    // partial view of i
 5) i.decision : value                        // decision value
 6) i.asked : set of nodes                    // nodes requiring the decision
 7) i.values : set of ⟨node, value⟩ tuples    // reported decisions

message:
 8) SET_DECISION:
 9)     decision : value                      // the decided value

** All Nodes **
Task MAIN
10) i.decision, i.in_the_sink ← ⊥;
11) i.values, i.known ← ∅;
12) i.asked ← Its value comes from Algorithm 2 (DISCOVERY)
13) Fork GATHER_DECISION;
14) (i.in_the_sink, i.known) ← SINK();

** Node In Sink **
15) if i.in_the_sink then // underlying Byzantine consensus
16)     Consensus.propose(i.initial)
17)     upon Consensus.decide(v)
18)     i.decision ← v;
19)     ∀j ∈ i.asked, send SET_DECISION(i.decision) to j;
20)     return i.decision;
21) end if

** Node Not In Sink **
Task GATHER_DECISION
22) upon receipt of SET_DECISION(v) from p
23) if i.decision = ⊥ then
24)     i.values ← i.values ∪ {⟨p, v⟩};
25)     if #⟨*,v⟩ i.values > f then
26)         i.decision ← v;
27)         return i.decision;
28)     end if
29) end if
```

Description. The algorithm starts with each process executing the SINK protocol (line 14) in order to get its system partial view and decide if it is in G_{sink}. If process $i \notin G_{sink}$, it could had been blocked on the execution of Algorithm 2 (DISCOVERY). Anyway, it will wait for a decision on the execution of the GATHER_DECISION *Task* that has been launched in line 13. If process $i \in G_{sink}$, it terminates SINK and can progress on the execution of the remaining of the algorithm. Thus, depending on whether or not the process belongs to G_{sink}, two distinct behaviors are possible:

(1) If $i \in G_{sink}$, it executes a standard Byzantine consensus (lines 16-17) with the processes in its view ($i.known$). We use the following interface to the standard consensus algorithm: Consensus.propose($value$) to initiate a consensus instance by proposing $value$ and Consensus.decide($decision$) that is a callback function called by the standard consensus algorithm to inform that a $decision$ was taken. When the decision is taken, it will send it to all processes that have been asked for it (line 19). These processes are in the $i.asked$ set and have been identified during Algorithm 2 (DISCOVERY). Notice that, during the execution of DISCOVERY, the decision could had already been taken, and, in this case, on the execution of the DELIVER *Task*, i will send it directly to asking processes.

(2) If $i \notin G_{sink}$, it does not participate in the standard consensus. During the execution of DISCOVERY, i has sent messages to all f-reachable participants. Since all processes in G_{sink} are f-reachable from i, it is ensured that each correct process $j \in G_{sink}$ sends SET_DECISION($j.decision$) to i when the decision has been taken, since moreover $i \in j.asked$. Node i decides for a value v only after it has received v from at least $f + 1$ other participants, ensuring that v is gathered from at least one correct participant (lines 22-29).

4.3.2 CONSENSUS Correctness

Theorem 1 shows that the CONSENSUS protocol solves BFT-CUP. However, before presenting this theorem, we need to prove the following lemma.

Lemma 11 *A correct node $j \in G_{di}$ that communicated with correct node $i \in G_{sink}$ before the decision has been taken in line 18 of CONSENSUS is in $i.asked$.*

Proof. From Lemmata 4 and 5, every message that j reachable broadcasts is delivered by i. Consequently, since the DELIVER *Task* of the DISCOVERY algorithm keeps executing, as soon as a message from j is delivered by i before the decision has been taken, it will put j in $i.asked$ (lines 15–17 of DISCOVERY). □

Theorem 1 *Under Assumptions 1 and 2, algorithm CONSENSUS solves BFT-CUP.*

Proof. Depending on whether or not node i belongs to G_{sink}, two distinct behaviors are possible:

(1) If $i \in G_{sink}$: On the execution of SINK (line 14), i gets $\langle true, V_{sink}\rangle$ (Lemma 9). Then, i executes an underlying standard Byzantine consensus (line 16) with the nodes in V_{sink}. From Assumption 2, V_{sink} has at least $2f + 1$ correct nodes, then all the properties of the underlying Byzantine consensus will be met, i.e., *Validity*, *Integrity*, *Agreement* and *Termination*. Thus, process i will eventually execute line 17 and decide. The decided value is then sent to all nodes in $i.asked$ (line 19). Finally, the decided value is returned to the application (line 20). From Lemma 11, every correct node $j \in G_{di}$ that communicates with i, $j \in i.asked$. However, if due to the lack of synchronism, the messages from $j \in G_{di}$ have not yet arrived at i on the time i is executing CONSENSUS, $j \notin i.asked$. In this case, since the DELIVER *Task* of the DISCOVERY algorithm keeps executing, as soon as these messages arrive at i, it will send the decision value to j (line 19 of DISCOVERY).

(2) If $i \notin G_{sink}$: On the execution of SINK (line 14), i can be blocked or otherwise can get $\langle false, i.known \rangle$ (Lemma 10). In any case, i will not participate to the standard consensus (line 15). It will expect for the decision on the execution of the GATHER_DECISION *Task* (lines 22–29) launched in line 13. When a SET_DECISION(v) arrives at i (line 22), if i has not yet decided, it will store v in the $i.values$ set (line 24) and, as soon as $f+1$ equal values are received (line 25), i returns the decided value to the application (line 27). From the behavior (1) above, there are at least $2f+1$ correct nodes in G_{sink} who sent the decision (when it is taken) to i, either on the execution of line 19 of the CONSENSUS or on the execution of line 19 of the DELIVER *Task* of DISCOVERY. This ensures that at least $f+1$ decision messages will arrive, eventually satisfying the predicate at line 25. This predicate avoids a collusion of f malicious participants and the process reliable decide, returning the decided value to the application (lines 26–27). This concludes the proof and the lemma follows. □

5 NECESSARY CONDITIONS TO BFT-CUP

This section presents the necessary conditions to solve BFT-CUP, namely, the knowledge connectivity model of Assumption 1 and the BFT consensus model of Assumption 2.

In the following lemmata, $G_{di} = (V, E)$ is the directed knowledge connectivity graph returned by a PD, G is the undirected graph obtained from G_{di} and $Dag(G_{di})$ is the DAG (Direct Acyclic Graph) obtained by reducing G_{di} to its strongly connected components.

We start by proving that G_{di} should not have more than one sink (Lemma 12), G should be connected (Lemma 13), and the knowledge connectivity defined by the safe Byzantine failure pattern (Assumption 1) is necessary to solve BFT-CUP (Lemma 14). Afterward, we prove that $2f+1$ correct processes in G_{sink} (Assumption 2) is also necessary to solve BFT-CUP (Lemma 15). Finally, Theorem 2 concludes the proof presenting the necessary and sufficient conditions for BFT-CUP.

Lemma 12 *In order to solve BFT-CUP in an asynchronous system extended with a PD, $Dag(G_{di})$ should have exactly one sink component.*

Proof. Assume for the purpose of contradiction, a standard proof technique, that $Dag(G_{di})$ obtained from $G_{di} \in$ PD has more than one sink, yet there exists a BFT-CUP protocol $\mathcal{A}$ in the asynchronous system. We will show that $\mathcal{A}$ admits an execution that violates *Agreement*.

Consider system $\mathcal{X}$ in which $Dag(G_{di})$ has more than one sink. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two of those sinks. Assume that all nodes in G_1 have input value equals to v_1 and all nodes in G_2 have input value equals to $v_2 \neq v_1$. Let us construct a system $\mathcal{X}_1$ derived from $\mathcal{X}$ composed only by processes in G_1 such that the initial input values of the processes are equal v_1 as well. Consider an execution e_1 of $\mathcal{A}$ for $\mathcal{X}_1$. By the *Termination* property, processes in G_1 eventually decide at time t_1. By the *Validity* property, they decide v_1. Similarly, it is possible to construct a system $\mathcal{X}_2$ derived from $\mathcal{X}$ composed only by processes in G_2 such that the initial input values of the processes are equal v_2 as well. Consider an execution e_2 of $\mathcal{A}$ for $\mathcal{X}_2$. By the *Termination* property, processes in G_2 eventually decide at time t_2. By the *Validity* property, they decide v_2.

Consider now the original system $\mathcal{X}$ containing all processes from G_1 and G_2. From the system assumptions, the cardinality n of the system composition is unknown and the only knowledge that a process has is provided by PD and represented by G_{di}. Let $K_1 = \bigcup_{i \in V_1} i.\text{PD}$ be the set of processes known by all processes in G_1. Similarly, let $K_2 = \bigcup_{j \in V_2} j.\text{PD}$ be the set of processes known by all processes in G_2. From the graph properties, there is no outgoing edge from a process in a sink to the other processes outside the sink. Thus, $K_1 \subseteq V_1$ and $K_2 \subseteq V_2$ and processes either in G_1 or G_2 have no knowledge about the other processes in G_{di} (including processes in the other sink components). Also, since the system is asynchronous, consider that a process i outside a sink, $i \in V \setminus \{V_1 \cup V_2\}$, does not take any step until time $t = max\{t_1, t_2\}$; or, alternatively, if i sends a message to a process $j \in V_1 \cup V_2$, the delivery of this message is delayed until after time t.

Clearly, it is possible to have an execution e of algorithm $\mathcal{A}$ in system $\mathcal{X}$ in which processes in G_1 take steps exactly as in execution e_1 for system $\mathcal{X}_1$ up to time t. In both executions, the steps that these processes take are the same up to time t. Then, in execution e, processes in G_1 decide for v_1 at $t_1 \leq t$. Similarly, in the same execution e of algorithm $\mathcal{A}$ in system $\mathcal{X}$, processes in G_2 may take steps exactly as in execution e_2 for system $\mathcal{X}_2$ up to time t. In both cases, the steps that these processes take in executions e_2 and e are the same steps up to time t. Then, in execution e, processes in G_2 decide for v_2 at $t_2 \leq t$. But, since processes in G_1 decide for v_1 and processes in G_2 decide for v_2, $v_1 \neq v_2$, the *Agreement* property is violated in execution e, thus reaching a contradiction that $\mathcal{A}$ solves BFT-CUP in system $\mathcal{X}$. □

Lemma 13 *In order to solve BFT-CUP in an asynchronous system extended with a PD, G should be connected.*

Proof. The proof follows directly from Lemma 12, since if G is not connected, there exists at least two sink components in $Dag(G_{di})$. □

Observation 3 *Following the results of Dolev [26], in an asynchronous unknown network, the number of malicious failures should be less than half of the connectivity degree in order to processes be able to communicate properly. This ensures the authentication of the communication: the receiver of some message is able to verify the identity of its sender, ensuring that no forged messages are processed. Without this, it is not possible to tolerate process misbehavior in an asynchronous system, since a single faulty process can play the roles of all other processes to others.*

Lemma 14 *Let us consider an asynchronous system with unknown participants prone to at most f Byzantine failures in which the BFT-CUP problem can be solved. Let $\mathcal{A}$ be a protocol able to solve BFT-CUP based on the PD information. Protocol $\mathcal{A}$ requires the knowledge connectivity graph G_{di} to satisfy the safe Byzantine failure pattern.*

Proof. The safe Byzantine failure pattern states that $G_{di} \setminus F \in k$-OSR, $k \geq f+1$, assuming that F is the set

of participants in G_{di} that actually fail, $|F| \leq f$. Let $G_{sink} = (V_{sink}, E_{sink})$ be the sink component of G_{di}. The conditions stated in the lemma ensure that whatever the actual pattern of failures, $G_{di} \setminus F$ satisfies the properties of the k-OSR PD, $k \geq f + 1$. As a result, there exists at least $k \geq f + 1$ node-disjoint paths composed by *correct processes* between processes in G_{sink} and between a process outside G_{sink} to a process inside it.

Now, assume by contradiction that there is a protocol $\mathcal{A}$ that solves BFT-CUP with a PD that does not satisfy the safe Byzantine failure pattern. The following four scenarios are possible: either (1) the undirected graph G obtained from $G_{di} \setminus F$ is not connected; or (2) the DAG obtained from the decomposition of $G_{di} \setminus F$ to its strongly connected components has more than one sink; or (3) the unique sink component G_{sink} of $G_{di} \setminus F$, is not k-strongly connected, $k \geq f + 1$ and thus, there exists less than $(f + 1)$-correct-node-disjoint paths between its processes; or (4) there are i, j, such that $i \notin G_{sink}$ and $j \in G_{sink}$ and there exists less than $(f + 1)$-node-disjoint paths from i to j in $G_{di} \setminus F$.

Scenario (1): Connectivity is a necessary condition to solve BFT-CUP (Lemma 13).

Scenario (2): One sink component is a necessary condition to solve BFT-CUP (Lemma 12).

Scenario (3): From Observation 3, G_{sink} does not have enough connectivity and the existence of f faulty nodes may split it into at least two components in G_{di}, G_1 and G_2, in a way that no message from nodes in G_1 (respectively, G_2) can be authenticated by nodes in G_2 (respectively, G_1). In this case, processes will believe that G_{di} has at least two sinks: G_1 and G_2. From Lemma 12, one sink component is necessary.

Scenario (4): if there exists less than $(f+1)$-node-disjoint paths from $i \notin G_{sink}$ to $j \in G_{sink}$ in $G_{di} \setminus F$, then i f-reaches at most f nodes in G_{sink} (since $G_{sink} \setminus F$ is $(f+1)$-strongly connected). Let $C \subset G_{di} \setminus F$ be the set of these nodes f-reachable from i; then, $|C| \leq f$ and $j \notin C$. Notice that C is a vertex cut of at most f processes in $G_{di} \setminus F$, dividing $G_{di} \setminus F$ into at least two components: BC (before cut) and AC (after cut), such that, $i \in BC$ and $G_{sink} = AC \cup C$.

Now, we will show that there is an execution e of protocol $\mathcal{A}$ that violates *Agreement*. As Π is unknown, to solve BFT-CUP, i needs to find other processes with which it can collaborate (a subset of Π). The only way to do that is by executing a search in G_{di}. Going on the search, i iteratively requests newly known processes about their view to get knowledge improvement. This search terminates when i discovers a sufficient number of processes in G_{di}. Since the system is asynchronous and at least f processes in G_{di} could be malicious, in order to ensure *Termination* the search has to end when i has inquired all known processes, except from f. Clearly, we could have an execution e of protocol $\mathcal{A}$ in which i finishes its search before inquiring the processes in C, since $|C| \leq f$. We can consider that in this execution e, i has previously discovered the processes in BC and then in C; thus, in the end of the search, i discovers $BC \cup C$. Since processes in C are not inquired by i, it will have no knowledge about AC. By generalization, in execution e, all processes in BC discovers $BC \cup C$ as well. Cleary, the execution e will exhibit two sinks: the actual $G_{sink} = AC \cup C$ and $BC \cup C$. By Lemma 12, one sink component is a necessary condition to ensure BFT-CUP Agreement.

From Scenarios 1 to 4, we conclude that $\mathcal{A}$ does not exists. □

Lemma 15 *In order to solve BFT-CUP in an asynchronous system extended with a PD, the sink component of G_{di} must have at least $2f + 1$ correct processes.*

Proof. A corollary of Lemma 12 is that decisions must be taken by the processes in the sink component of G_{di} and, in order to ensure *Agreement*, the non-sink participants should wait for the decision coming from the sink processes. According to [5], a necessary condition to solve standard Byzantine consensus in a non-synchronous system is the existence of at least $2f + 1$ correct processes in the system. Consequently, the lemma follows. □

Theorem 2 *Let us consider an asynchronous system with unknown participants prone to at most f Byzantine failures in which the BFT-CUP problem can be solved. Let $\mathcal{A}$ be a protocol able to solve BFT-CUP based on the PD information.*

- NECESSITY: *Protocol $\mathcal{A}$ requires G_{di} to follow the safe Byzantine failure pattern (Assumption 1) and the unique sink component of G_{di} to have at least $2f + 1$ correct processes (Assumption 2).*
- SUFFICIENCY: *The safe Byzantine failure pattern (Assumption 1) and $2f + 1$ correct processes in the sink (Assumption 2) are sufficient for protocol $\mathcal{A}$ be able to solve BFT-CUP.*

Proof. The necessity follows directly from Lemmata 14 and 15. The sufficiency follows directly from Theorem 1. □

On one hand, the sufficient conditions specify what is enough for solving BFT-CUP (but it does not mean that all of these conditions are necessary). On the other hand, the necessary conditions specify minimum requirements to solve BFT-CUP (but it does not mean that they are sufficient). This paper proves that the *safe Byzantine failure pattern* together with *2f+1 correct sink participants* are both sufficient and necessary to solve BFT-CUP (Theorem 2).

6 CONCLUSION

In this paper, we identified necessary and sufficient conditions to solve the BFT-CUP problem in an asynchronous system. These conditions are related with the degree of knowledge about the system composition that participants must initially obtain. The proposed protocols complement previous works about consensus with unknown participants by decreasing the minimum degree of knowledge necessary to solve BFT-CUP. The new threshold is showed to be optimal. As a side effect, a BFT dissemination primitive, namely *reachable reliable broadcast*, has been defined and can be used in other protocols for unknown networks.

REFERENCES

[1] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[2] M. Correia, N. F. Neves, and P. Veríssimo, "From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures," *The Computer Journal*, vol. 49, no. 1, 2006.

[3] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[4] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[5] S. Toueg, "Randomized Byzantine Agreements," in *Proc. of 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 163–178.

[6] L. Lamport, "The part-time parliament," *ACM Transactions Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[7] F. B. Schneider, "Implementing fault-tolerant service using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[8] C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of ACM*, vol. 35, no. 2, pp. 288–322, 1988.

[9] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems," *IEEE Trans. on Dep. and Secure Computing*, vol. 2, no. 1, pp. 46–56, 2005.

[10] J.-P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Trans. on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[11] O. Rutti, Z. Milosevic, and A. Schiper, "Generic construction of consensus algorithms for benign and Byzantine faults," in *40th Int. Conf. on Dependable Systems and Networks*, 2010, pp. 343 –352.

[12] D. Cavin, Y. Sasson, and A. Schiper, "Consensus with unknown participants or fundamental self-organization," in *Proc. of 3rd Int. Conf. on Ad hoc Networks and Wireless*, 2004, pp. 135–148.

[13] ——, "Reaching agreement with unknown participants in mobile self-organized networks in spite of process crashes," EPFL - LSR, Tech. Rep. IC/2005/026, 2005.

[14] F. Greve and S. Tixeuil, "Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2007.

[15] ——, "Conditions for the solvability of fault-tolerant consensus in asynchronous unknown networks: Invited paper," in *III Workshop on Reliability, Availability, and Security. Co-located with the 29th Symposium on Principles of Distributed Computing*, 2010.

[16] E. A. P. Alchieri, A. N. Bessani, J. da Silva Fraga, and F. Greve, "Byzantine consensus with unknown participants," in *12th Int. Conf. On Principles Of DIstributed Systems*, 2008, pp. 22–40.

[17] C. Khouri, F. Greve, and S. Tixeuil, "Consensus with unknown participants in shared memory," in *32th IEEE International Symposium on Reliable Distributed Systems*, 2013.

[18] J. Chalopin, E. Godard, and A. Naudin, "What do we need to know to elect in networks with unknown participants?" in *21th International Colloquium on Structural Information and Communication Complexity*, 2014.

[19] A. K. Datta, L. L. Larmore, S. Devismes, F. Kawala, and M. Potop-Butucaru, "Multi-resource allocation with unknown participants," in *International Conference on Computing, Networking and Communications (ICNC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 200–206.

[20] F. Greve, P. Sens, L. Arantes, and V. Simon, "Eventually strong failure detector with unknown membership," *The Computer Journal*, vol. 55, no. 12, pp. 1507–1524, Dec. 2012.

[21] J. Douceur, "The sybil attack," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[22] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens, "ODSBR: An on-demand secure Byzantine resilient routing protocol for wireless ad hoc networks," *ACM Transactions on Information and Systems Security*, vol. 10, no. 4, Jan. 2008.

[23] P. Kotzanikolaou, R. Mavropodi, and C. Douligeris, "Secure multipath routing for mobile ad hoc networks," in *Proc. of 2nd Annual Wireless On-demand Network Systems and Services*, 2005.

[24] G. Bracha, "An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol," in *Proc. of the 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 154–162.

[25] H. Moniz, N. Neves, and M. Correia, "Byzantine fault-tolerant consensus in wireless ad hoc networks," *IEEE Transactions on Mobile Computing*, vol. 12, no. 12, pp. 2441–2454, 2013.

[26] D. Dolev, "The Byzantine generals strike again," *Journal of Algorithms*, no. 3, pp. 14–30, 1982.

Eduardo Adilio Pelinson Alchieri is currently an Adjunct Professor at the Department of Computer Science of the University of Brasília, Brazil. He received his B.S. degree in Computer Science from Federal University of Santa Catarina (UFSC), Brazil in 2004, the MSE in Electrical Engineering from UFSC in 2007 and the PhD in Automation and Systems Engineering from UFSC in 2011. His main interests are distributed algorithms, fault tolerance, dynamic distributed systems, self organizing systems, middleware and systems architecture. He has received financial support from research agencies (CNPq and CAPES).

Alysson Bessani is an Associate Professor of the Department of Informatics of the University of Lisboa Faculty of Sciences, Portugal, and a member of LASIGE research unit and the Navigators research team. He received his B.S. degree in Computer Science from Maringá State University, Brazil in 2001, the MSE in Electrical Engineering from Santa Catarina Federal University (UFSC), Brazil in 2002 and the PhD in Electrical Engineering from the same university in 2006. His main interests are distributed algorithms, Byzantine fault tolerance, coordination, middleware and systems architecture. More information about him is available at http://www.di.fc.ul.pt/~bessani.

Fabíola Greve is currently an Associate Professor in the Department of Computer Science at the Federal University of Bahia, Brazil, where she acts as the leader of the distributed computing group Gaudi. She received a PhD degree in computer science in 2002 from Rennes University, INRIA Labs, France and a Master degree in computer science in 1991 from UNICAMP, Brazil. Her main interests span the domains of distributed computing and fault tolerance and her current projects aim at identifying conditions and protocols able to provide fault tolerance in dynamic and self organizing systems. She has published a number of scientific papers in these areas and she's been serving as principal investigator of funded research projects and as a program committee member of some of the main conferences and journals in the domain. She also has received financial support from research agencies including CNPq and CAPES.

Joni da Silva Fraga received the B.S. degree in Electrical Engineering in 1975 from Federal University of Rio Grande do Sul (UFRGS), the MSE degree in Electrical Engineering in 1979 from the Federal University of Santa Catarina (UFSC), and the PhD degree in Computing Science (Docteur de l'INPT/LAAS) from the Institut National Polytechnique de Toulouse, France, in 1985. Also, he was a visiting researcher at UCI (University of California, Irvine) in 1992-1993. Since 1977 he has been a faculty member and later a Professor in the Department of Automation and Systems at UFSC, in Brazil. His research interests are centered on Distributed Systems, with activities mainly in the following topics: Security, Intrusion Tolerance, Fault Tolerance, Distributed Algorithms and Middleware. He has been coordinator and investigator of the GCSeg research group. He also has received financial support from research agencies including CNPq, CAPES, and RNP. Prof. Fraga has over 200 scientific publications and is a Member of the IEEE and of Brazilian scientific societies.