Byzantine Consensus with Unknown Participants

Eduardo A. P. Alchieri[1], Alysson Neves Bessani[2],
Joni da Silva Fraga[1], and Fabíola Greve[3]

[1] Department of Automation and Systems
Federal University of Santa Catarina (UFSC)
Florianópolis, SC - Brazil
alchieri@das.ufsc.br,fraga@das.ufsc.br
[2] Large-Scale Informatics Systems Laboratory
Faculty of Sciences, University of Lisbon
Lisbon, Portugal
bessani@di.fc.ul.pt
[3] Department of Computer Science
Federal University of Bahia (UFBA)
Bahia, BA - Brazil
fabiola@dcc.ufba.br

Abstract. Consensus is a fundamental building block used 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 classical networks, few studies have been conducted in order to solve it in the context of dynamic and self-organizing systems characterized by unknown networks. While in a classical network the set of participants is static and known, in a scenario of unknown networks, the set and number of participants are previously unknown. This work goes one step further and 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 can behave maliciously. This paper presents a solution for BFT-CUP that does not require digital signatures. The algorithms are shown to be optimal in terms of synchrony and knowledge connectivity among participants in the system.

Keywords: Consensus, Byzantine fault tolerance, Self-organizing systems.

1 Introduction

The *consensus* problem [1,2,3,4,5], and more generally the *agreement* problems, form the basis of almost all solutions related to the development of reliable distributed systems. Through these protocols, participants are able to coordinate their actions in order to maintain state consistency and ensure system progress. This problem has been extensively studied in classical 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 [2].

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 22–40, 2008.

© Springer-Verlag Berlin Heidelberg 2008

In self-organizing systems, such as wireless mobile ad-hoc networks, sensor networks and, in a different context, unstructured peer to peer networks (P2P), solving consensus is even more difficult. In these environments, an initial knowledge about participants in the system is a strong assumption to be adopted and the number of participants and their knowledge cannot be previously determined. These environments define indeed a new model of distributed systems which has essential differences regarding the classical one. Thus, it brings new challenges to the specification and resolution of fundamental problems. In the case of consensus, the majority of existing protocols are not suitable for the new dynamic model because their computation model consists of a set of initially known nodes. The only notably exceptions are the works of Cavin *et al.* [6,7] and Greve *et al.* [8].

Cavin *et al.* [6,7] defined a new problem named FT-CUP (*fault-tolerant consensus with unknown participants*) which keeps the consensus definition but assumes that nodes are *not* aware of Π, the set of processes in the system. They identified necessary and sufficient conditions in order to solve FT-CUP concerning knowledge about the system composition and synchrony requirements regarding the failure detection. They concluded that in order to solve FT-CUP in a scenario with the weakest knowledge connectivity, the strongest synchrony conditions are necessary, which are represented by failures detectors of the class $\mathscr{P}$ [4].

Greve and Tixeuil [8] 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 classical environment, which are represented by failure detectors of the class $\diamond\mathscr{S}$ [4]. The approach followed on the design of their FT-CUP protocol is modular: Initially, algorithms identify a set of participants in the network that share the same view of the system. Subsequently, any classical consensus – like for example, those initially designed for traditional networks – can be reused and executed by these participants.

Our work extends these results and study the problem of *Byzantine Fault-Tolerant Consensus with Unknown Participants* (BFT-CUP). This new problem aims at solving CUP in unknown networks with the additional requirement that participants in the system can behave maliciously [1]. The main contribution of the paper is then the identification of necessary and sufficient conditions in order to solve BFT-CUP. More specifically, an algorithm for solving BFT-CUP is presented for a scenario which does not require the use of digital signatures (a major source of performance overhead on Byzantine fault-tolerant protocols [9]). Finally, we show that this algorithm is optimal in terms of synchrony and knowledge connectivity requirements, establishing then the necessary and sufficient conditions for BFT-CUP solvability in this context.

The paper is organized in the following way. Section 2 presents our system model and the concept of participant detectors, among other preliminary definitions used in this paper. Section 3 describes a basic dissemination protocol used for process communication. BFT-CUP protocols and respective necessary and sufficient proofs are described in Section 4. Section 5 presents some comments about our protocol. Section 6 presents our final remarks.

2 Preliminaries

2.1 System Model

We consider a distributed system composed by a finite set Π of n processes (also called participants or nodes) drawn from a larger universe U. In a *known network*, Π and n is known to every participanting 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* [1], i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. Processes that do not follow their algorithm in some way are 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, it does know the expected maximum number of process that may fail, denoted by f. Moreover, we assume that all processes have a unique id, and that it is infeasible for a faulty process to obtain additional ids to be able to launch a *sybil attack* [10] against the system.

Processes communicate by sending and receiving messages through *authenticated and reliable point to point channels* established between known processes[1]. Authenticity of messages disseminated to a not yet known node 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 [11,12,13], 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. For example, [12] presents a secure multipath routing protocol that guarantees a proper communication between two processes provided that there is at least one path between these processes that is not compromised, i.e., none of its processes or channels are faulty.

There are no assumptions on the relative speed of processes or on message transfer delays, i.e., the system is asynchronous. However, the protocol presented in this paper uses an underlying classical Byzantine consensus that could be implemented over an eventually synchronous system [14] (e.g., Byzantine Paxos [9]) or over a completely asynchronous system (e.g., using a randomized consensus protocol [5,15,16]). Thus, our protocol requires the same level of synchrony required by the underlying classical Byzantine consensus protocol.

2.2 Participant Detectors

To solve any nontrivial distributed problem, processes must somehow get a partial knowledge about the others if some cooperation is expected. The *participant detector* oracle, namely PD, was proposed to handle this subset of known processes [6]. 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

[1] Without authenticated channels 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 some (victim) process.

queried by i, $i.PD$ returns a subset of processes in Π with whom i can collaborate. Let $i.PD(t)$ be the query of i at time t. The information provided by $i.PD$ can evolve between queries, but must satisfy the following two properties:

- *Information Inclusion*: The information returned by the participant detectors is non-decreasing over time, i.e., $\forall i \in \Pi, \forall t' \geq t : i.PD(t) \subseteq i.PD(t')$;
- *Information Accuracy*: The participant detectors do not make mistakes, i.e., $\forall i \in \Pi, \forall t : i.PD(t) \subseteq \Pi$.

Participant detectors provide an initial context about participants present in the system by which it is possible to expand the knowledge about Π. Thus, the participant detector abstraction enriches the system with a knowledge connectivity graph. This graph is directed since the knowledge provided by participant detectors is not necessarily bidirectional [6].

Definition 1. ***Knowledge Connectivity Graph***: *Let $G_{di} = (V, \xi)$ be the directed graph representing the knowledge relation determined by the PD oracle. Then, $V = \Pi$ and $(i, j) \in \xi$ iff $j \in i.PD$, i.e., i knows j.*

Definition 2. ***Undirected Knowledge Connectivity Graph***: *Let $G = (V, \xi)$ be the undirected graph representing the knowlegde relation determined by the PD oracle. Then, $V = \Pi$ and $(i, j) \in \xi$ iff $j \in i.PD$ or $i \in j.PD$, i.e., i knows j or j knows i.*

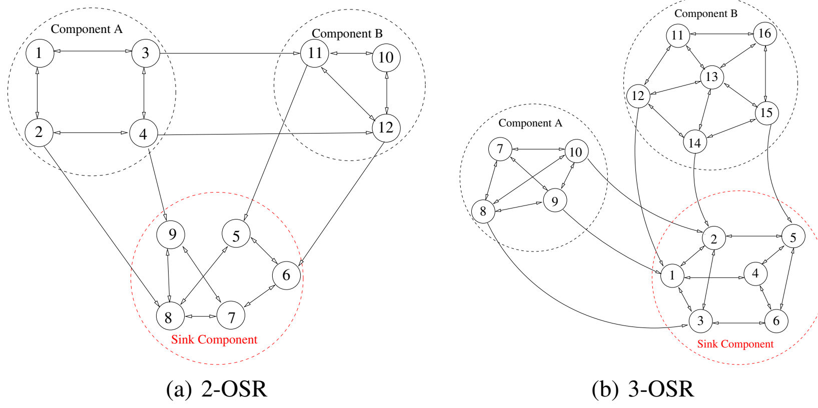
Based on the properties of the knowledge connectivity graph, some classes of participant detectors have been proposed to solve CUP [6] and FT-CUP [7,8]. Before defining how a participant detector encapsulates the knowledge of a system, let us define some graph notations. We say that a component G_c of G_{di} is *k-strongly connected* if for any pair (v_i,v_j) of nodes in G_c, v_i can reach v_j through k node-disjoint paths. A component G_s of G_{di} is a *sink component* when there is no path from a node in G_s to other nodes of G_{di}, except nodes in G_s itself. In this paper we use the weakest participant detector defined to solve FT-CUP, which is called k-OSR [8].

Definition 3. ***k-One Sink Reducibility (k-OSR) PD***: *The knowledge connectivity graph G_{di}, which represents the knowledge induced by PD, 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.*

To better illustrate Definition 3, Figure 1 presents two graphs G_{di} induced by a k-OSR participant detector. Figures 1(a) and 1(b) show knowledge relations induced by participant detectors of the class 2-OSR and 3-OSR, respectively. For example, in Figure 1(a), the value returned by $1.PD$ is the subset $\{2,3\} \subset \Pi$.

In our algorithms, we assume that for each process i, its participant detector $i.PD$ is queried exactly once at the beginning of the protocol execution. This can be implemented by caching the result of the first query to $i.PD$ and returning that value in



(a) 2-OSR (b) 3-OSR

Fig. 1. Knowledge Connectivity Graphs Induced by k-OSR Participant Detectors

subsequent calls. This ensures that the partial view about the initial composition of the system is consistent for all nodes in the system, what defines a common knowledge connectivity graph G_{di}. Also, in this work we say that some participant p is *neighbor* of another participant i iff $p \in i.PD$.

2.3 The Consensus Problem

In a distributed system, the consensus problem consists of ensuring that all correct processes eventually decide the same value, previously proposed by some processes in the system. 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 [4]:

- *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 *Byzantine Fault-Tolerant Consensus with Unknown Participants*, namely BFT-CUP, proposes to solve consensus in unknown networks with the additional requirement that a bounded number of participants in the system can behave maliciously.

3 Reachable Reliable Broadcast

This section introduces a new primitive, namely *reachable reliable broadcast*, used by processes of the system to communicate. It is invoked by two basic operations:

- ***reachable_send***(m,p) – through which the participant p sends the message m to all reachable participants from p. A participant q is reachable from another participant

[2] If a randomized protocol such as [5,15,17] is used as an underlying Byzantine consensus, the termination is ensured only with probability 1.

p if there is enough connectivity from p to q (see below). In this case, q is a receiver of messages disseminated by p.
- ***reachable_deliver(m,p)*** – invoked by the receiver to deliver a message m disseminated by the participant p.

This primitive should satisfy the following four properties:

- *Validity*: If a correct participant p disseminates a message m, then m is eventually delivered by a correct participant reachable from p or there is no correct participant reachable from p;
- *Agreement*: If a correct participant delivers some message m, disseminated by a correct participant p, then all correct participants reachable from p eventually deliver m;
- *Integrity*: For any message m, every correct participant p delivers m only if m was previously disseminated by some participant p', in this case p is reachable from p'.

Notice that these properties establish a communication primitive with specification similar to the usual reliable broadcast [4,5,15]. Nonetheless, the proposed primitive ensures the delivery to all correct processes reachable in the system.

Implementation. The main idea of our implementation is that participants execute a flood of their messages to all reachable processes, which, in turn, will deliver these messages as soon as its authenticity has been proved. Assuming a k-OSR PD, a participant q is reachable from a participant p if there is enough connectivity in the knowlegde graph, i.e., if there are at least $2f+1$ node-disjoint paths from p to q ($k \geq 2f+1$). This connectivity is necessary to ensure that all reachable processes will be able to receive and authenticate messages.

In our implementation, formally described in Algorithm 1, a process i disseminates a message m through the system by executing the procedure *reachable_send*. In this procedure (line 6), i sends m to its neighbors (i.e., processes in $i.PD$) and when m is received at some process p, p forwards m to its neighbors and so on, until that m arrives at all reachable participants (line 17). Moreover, p stores m together with the route traversed by m in a buffer (line 11). Also, p delivers m if it has received m through $f+1$ node-disjoint paths (lines 13-14), i.e., the authenticity of m has been verified. Afterward, since m has been delivered, p removes it from the buffer of received messages (line 15). The function $computeRoutes(m.message, i.received_msgs)$ computes the number of node-disjoint paths through which $m.message$ has been received at participant i.

An important feature of this dissemination is that each message has the accumulated route according with the path traversed from the sender to some destination. 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 (line 8). This solution is based on the approach used in [18] and it enforces that each participant appends itself at the end of the routing information in order to send or forward a message. Nonetheless, a malicious participant is able to modify the accumulated route (removing or adding participants) and modify or block the message being propagated. Notice, however, that the connectivity of the knowledge graph ($k \geq 2f+1$) ensures that messages will be received at all reachable participants. Moreover, since a process delivers a message only

Algoritm 1. Dissemination algorithm executed at participant i.

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

variables:
 2. i.received_msgs : set of ⟨message, route⟩ tuples          // set of received messages

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

** Initiator Only **
procedure: reachable_send(message, sender)                          // sender = i
 6. ∀j ∈ i.PD, send REACHABLE_FLOODING(message, sender) to j;

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

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

after it has been received through $f+1$ node disjoint paths, it is able to verify its authenticity. These measures prevent the delivery of forged messages (generated by malicious participants), because the authenticity of them cannot be verified by correct processes.

An "undesirable" property of the proposed solution is that the same message, sent by some participant, could be delivered more than once by its receivers. This property does not affect the use of this protocol in our consensus protocol (Section 4). Thus, we do not deal with this limitation of the algorithm. However, it can be easily solved by using buffers to store delivered messages that must have unique identifiers.

Additionaly, each message' receiver, disseminated by some participant p, is able to send back a reply to p using some routing protocol resilient to Byzantine failures [11,12,13]. Our BFT-CUP protocol (Section 4) uses this algorithm to disseminate messages.

Sketch of Proof. The correctness of this protocol is based on the proof of the properties defined for the reachable reliable broadcast.

Validity: By assumption, the connectivity of the system is $k \geq 2f+1$. Thus, according to Definition 3, there are at least $2f+1$ node-disjoint paths from the sender of a message m to the receivers (nodes that are reachable from the sender). Moreover, as validity is established over messages sent by correct participants (correct sender), there are at least $f+1$ node-disjoint paths formed only by correct participants, through which it is guaranteed that the same message m will reach the correct receivers. In this case, the predicate of line 8 will be true at least $f+1$ times and the authenticity of m can be verified through redundancy. This is done by the execution of lines 9–12, which are responsible to maintain information regarding the different routes from which m has been received. Whenever the message authenticity is proved, i.e., m has been received by at least $f+1$ different routes (line 13), the delivery of m is authorized by the invocation of *reachable_deliver* (line 14).

Agreement: As the agreement is established over messages sent by correct participants, this proof is identical to the validity proof.

Integrity: A message is delivered only after its reception through $f+1$ node-disjoint paths (lines 13-14), what guarantees that this message is authentic, i.e., this message was really sent by its sender (*sender*). Thus, a malicious participant j is not able to forge that message m was sent by a participant i because the autenticity of m will not be proven. That is, a receiver r will not be able to find $f+1$ node-disjoint paths from i to r through which m has been received. Even with a collusion of up to f malicious participants, r will obtain at most f node-disjoint paths through which m was received "from i" (each of these f paths could contain one malicious participant). □

4 BFT-CUP: Byzantine Consensus with Unknown Participants

This section presents our solution for BFT-CUP. Our protocol is based on the dissemination algorithm presented in Section 3, which, together with the underlying routing layer resilient to Byzantine failures, hides all details related to participants communication. Thereafter, as in [8], the consensus protocol with unknown participants is divided into three phases. In the first phase – called *participants discovery* (Section 4.1) – each participant increases its knowledge about other processes in the system, discovering the maximum possible number of participants that are present in some computation. The second phase – called *sink component determination* (Section 4.2) – defines which participants belong to the sink component of the knowlegde graph induced by a k-OSR PD. Thus, each participant will be able to determine whether it belongs to the sink component or not. In the last phase (Section 4.3), members of the sink component *execute a classical Byzantine fault tolerant consensus* and disseminate the decision value to other participants in the system. The number of participants in the sink component, namely n_{sink}, should be enough in order to e xecute a classical Byzantine fault-tolerant consensus. Usually $n_{sink} \geq 3f+1$, to run, for example, Byzantine Paxos [9,19].

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. Notice that, through

its local participant detector, a process is able to get an initial knowledge about the system that is not enough to solve BFT-CUP. Then, a process expands this knowledge by executing the DISCOVERY protocol, presented in Algorithm 2. The main idea is that each participant i broadcasts a message requesting information about neighbors of each reachable participant, making a sort of breadth-first search in the knowledge graph. At the end of the algorithm, i obtains the maximal set of reachable participants, which represents the participants known by i (a partial view of the system).

The algorithm uses three sets:

1. *i.known* – set containing identifiers of all processes known by i;
2. *i.msg_pend* – this set contains identifiers 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;
3. *i.nei_pend* – this set contains identifiers of processes that i knows, but does not know all of their neighbors (i is still waiting for information about them), i.e., for each $\langle j, j.neighbor \rangle \in i.nei_pend$, i knows j but does not know all neighbors of j.

In the initialization phase of the algorithm for participant i, the set *i.known* is updated to itself plus its neighbors, returned by *i.PD*, and the set *i.msg_pend* to its neighbors (line 7). Moreover, a message requesting information about neighbors is disseminated to all participants reachable from i (line 8). When a participant p delivers this message, p sends back to i a reply indicating its neighbors (line 9).

Upon receipt of a reply at participant i, the set of known participants is updated, along with the set of pending neighbors[3] and the set of pending messages (lines 10 - 12). The next step is to verify whether i has acquired knowledge about any new participant (line 13 - 16). Thus, i gets to know other participant j if at least $f+1$ other processes known by i reported to i that j is their neighbor (line 13). After this verification, the set of pending neighbors is updated (lines 17 - 21), according to the new participants discovered.

To determine if there is still some participant to be discovered, i uses the sets *i.msg_pend* and *i.nei_pend*, which store the pendencies related to the replies received by i. Then, the algorithm ends when there remain at most f pendencies (lines 22 - 24). The intuition behind this condition is that if there are at most f pendencies at process i, then i already has discovered all processes reachable from it because $k \geq 2f+1$. Thus, the algorithm ends by returning the set of participants discovered by i (line 23), which contains all participants (correct or faulty) reachable from it. Algorithm 2 satisfies some properties that are stated by Lemma 1.

Lemma 1. *Consider G_{di} a knowlegde graph induced by a k-OSR PD. Let $f < \frac{k}{2} < n$ be the number of nodes that may fail. Algorithm DISCOVERY executed by each correct participant p satisfies the following properties:*

- *Termination: p terminates the execution of the algorithm and returns a set of known processes;*
- *Accuracy: the algorithm returns the maximal set of processes reachable from p in G_{di}.*

[3] If i reaches p, i also reaches all neigbours of p and should receive a reply to its initial dissemination (line 8) from all of them.

Algorithm 2. Algorithm DISCOVERY executed at participant i.

constant:
1. f : *int* // upper bound on the number of failures

variables:
2. *i.known* : set of nodes // set of known nodes
3. *i.nei_pend* : set of $\langle node, node.neighbor \rangle$ tuples // i does not know all neighbors of *node*
4. *i.msg_pend* : set of nodes // nodes that i is waiting for messages (replies)

message:
5. SET_NEIGHBOR: // struct of the message SET_NEIGHBOR
6. *neighbor* : set of nodes // neighbors of the node that is sending the message

**** All Nodes ****
INIT:
7. $i.known \leftarrow \{i\} \cup i.PD$; $i.nei_pend \leftarrow \varnothing$; $i.msg_pend \leftarrow i.PD$;
8. $reachable_send(\text{GET_NEIGHBOR}, i)$;

upon execution of $reachable_deliver(\text{GET_NEIGHBOR}, sender)$
9. **send** $\text{SET_NEIGHBOR}(i.PD)$ to *sender*;

upon receipt of $\text{SET_NEIGHBOR}(m.neighbor)$ **from** *sender*
10. $i.known \leftarrow i.known \cup \{sender\}$;
11. $i.nei_pend \leftarrow i.nei_pend \cup \{\langle sender, m.neighbor \rangle\}$;
12. $i.msg_pend \leftarrow i.msg_pend \setminus \{sender\}$;
13. **if** $(\exists j : \#_{\langle *, \langle j \rangle \rangle} i.nei_pend > f) \wedge (j \notin i.known)$ **then**
14. $\quad i.known \leftarrow i.known \cup \{j\}$;
15. $\quad i.msg_pend \leftarrow i.msg_pend \cup \{j\}$;
16. **end if**
17. **for all** $\langle j, j.neighbor \rangle \in i.nei_pend$ **do**
18. $\quad$ **if** $(\forall z \in j.neighbor : z \in i.known)$ **then**
19. $\quad\quad i.nei_pend \leftarrow i.nei_pend \setminus \{\langle j, j.neighbor \rangle\}$;
20. $\quad$ **end if**
21. **end for**
22. **if** $(|i.nei_pend| + |i.msg_pend|) \leq f$ **then**
23. $\quad$ **return** *i.known*;
24. **end if**

Sketch of Proof. *Termination:* In the worst case, the algorithm ends when p receives replies from at least all correct reachable participants (line 22). By dissemination protocol properties, even in the presence of $f < \frac{k}{2}$ failures, all messages disseminated by p is delivered by its correct receivers (processes reachable from p). Thus, each correct participant reachable from p receives a request (line 8) and sends back a reply (line 9) that is received by p (lines 10 - 24). Then, as Π is finite, it is guaranteed that p receives replies from at least all correct reachable participants and ends the algorithm by returning a set of known processes.

Accuracy: The algorithm only ends when there remain at most f pendencies, which may be divided between processes that supply information about neighbors that do not

exist in the system (*i.nei_pend*) and processes from which p is still waiting for their messages/replies (*i.msg_pend*). Moreover, each participant z (being z reachable from p) is neighbor of at least $2f+1$ other participants, because $f < \frac{k}{2} < n$. Now, we have to consider two cases:

- If z is malicious and does not send back a reply to p (line 9), then p computes messages (replies) from at least $f+1$ correct neighbors of z, discovering z (lines 13 - 16).
- If z is correct, in the worst case, the message from z to p is delayed and f neighbors of z are malicious and do not inform p that z is in the system. However, as $f < \frac{k}{2}$, there remain $f+1$ correct neighbors of z in the system that inform p about the presence of z in the system.

As the algorithm only ends when there remain at most f pendencies, in both cases it is guaranteed that p only ends after discovering z, even if it firstly computes messages from the f malicious processes. □

4.2 Sink Component Determination

The objective of this phase is to define which participants belong to the sink component of the knowlegde graph induced by a k-OSR PD. More specifically, through Algorithm 3 (SINK), each participant is able to determine whether or not it is member of the sink component. The idea behind this algorithm is that after the execution of the procedure DISCOVERY, members in the sink component obtain the same partial view of the system, whereas in the other components, nodes have strictly more knowledge than in the sink, i.e., each node knows at least members of the component to which it belongs and members of the sink (see Definition 3).

In the initialization phase of the algorithm for participant i, i executes the DISCOVERY procedure in order to obtain its partial view of the system (line 8) and sends this view to all reachable/known participant (line 10). When these messages are delivered by some participant j, j sends back an *ack* response to i if it has the same knowledge of i (i.e., j belongs to the same component of i). Otherwise, j sends back a *nack* response (lines 11-15).

Upon receipt of a reply (lines 16-27), i updates the set of processes that have already answered (line 16). Moreover, if the reply received is a *nack*, the set of processes that belong to other components (*i.nacked*) is updated (line 18) and if the number of processes that do not belong to the same component of i is greater than f (line 19), i concludes that it does not belong to the sink component (lines 20-21). This condition holds because the system has at least $3f+1$ processes in the sink, known by all participants, that have strictly less knowledge about Π than processes not in the sink (Lemma 1). On the other hand, if i has received replies from all known processes, excluding f possible faulty (line 24), and the number of processes that belong to other components is not greater than f, i concludes that it belongs to the sink component (lines 25-26). This condition holds because processes in the sink receive messages only from members of this component. Moreover, in both cases, a collusion of f malicious participants cannot lead a process to decide incorrectly. Lemma 2 states the properties satisfied by Algorithm 3.

Algorithm 3. Algorithm SINK executed at participant *i*.

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

variables:
 2. i.known : set of nodes                                          // set of known nodes
 3. i.responded : set of nodes                   // set of nodes that has sent a reply to i
 4. i.nacked : set of nodes               // set of processes not in the same component of i
 5. i.in_the_sink : boolean                                            // is i in the sink?

message:
 6. RESPONSE:                                        // struct of the message RESPONSE
 7.      ack/nack : boolean

** All Nodes **
INIT:
 8. i.known ← DISCOVERY();
 9. i.responded ← {i}; i.nacked ← ∅;
10. reachable_send(i.known, i);

upon execution of reachable_deliver(sender.known, sender)
11. if i.known = sender.known then
12.     send RESPONSE(ack) to sender;
13. else
14.     send RESPONSE(nack) to sender;
15. end if

upon receipt of RESPONSE(m) from sender
16. i.responded ← i.responded ∪ {sender}
17. if m.nack then
18.     i.nacked ← i.nacked ∪ {sender};
19.     if |i.nacked| ≥ f + 1 then
20.         i.in_the_sink ← false;
21.         return ⟨i.in_the_sink, i.known⟩;
22.     end if
23. end if
24. if |i.responded| ≥ |i.known| − f then
25.     i.in_the_sink ← true;
26.     return ⟨i.in_the_sink, i.known⟩;
27. end if
```

Lemma 2. *Consider a k-OSR PD. Let $f < \frac{k}{2} < n$ be the number of nodes that may fail. Algorithm SINK, executed by each correct participant p of the system that has at least $3f+1$ nodes in the sink component, satisfies the following properties:*

- *Termination: p terminates the execution by deciding whether it belongs (true) or not (false) to the sink;*
- *Accuracy: p is in the unique k-strongly connected sink component iff algorithm SINK returns true.*

Sketch of Proof. *Termination:* For each participant p, the algorithm returns in two cases: (i) when it receives $f+1$ replies from processes that belong to other components (processes not in the sink – line 19) or (ii) when it receives replies from at least all correct known processes (processes in the sink – line 24). By properties of the dissemination protocol, even in the presence of $f < \frac{k}{2}$ failures, all messages disseminated by p are delivered by its receivers (processes reachable from p). Thus, each correct participant known by p (reachable from p) receives the request (line 10) and sends back a reply (lines 11-15) that is received by p (lines 16-27). Then, it is guaranteed that either (i) or (ii) always occur.

Accuracy: By Lemma 1, after execution of the DISCOVERY algorithm, each correct participant discovers the maximal set of participants reachable from it. Then, by Lemma 1 and by k-OSR PD properties, it is guaranteed that all correct processes that belong to the same component obtain the same partial view of the system. Thus, as members in the sink component receive replies only from members of this component, it is guaranteed that these participants end correctly (line 26). Moreover, as the sink has at least $3f+1$ nodes, members in other components know at least $2f+1$ correct members in the sink (Lemma 1). Then, before making a wrong decision, these members must compute at least $f+1$ replies from correct members in the sink (that have strictly less knowledge about Π, due to Lemma 1), what makes it possible for correct members not in the sink to end correctly (line 21). □

4.3 Achieving Consensus

This is the last phase of the protocol for solving BFT-CUP. Here, the main idea is to make members of the sink component execute a classical Byzantine consensus and send the decision value to other participants of the system. The optimal resilience of these algorithms to solve a classical consensus is $3f+1$ [3,9]. Thus, it is necessary at least $3f+1$ participants in the sink component.

The Algorithm 4 (CONSENSUS) presents this protocol. In the initialization, each participant executes the SINK procedure (line 11) in order to get its partial view of the system and decide whether or not it belongs to the sink component. Depending on whether or not the node belongs to the sink, two distinct behaviors are possible:

1. Nodes in the sink execute a classical consensus (line 13) and send the decision value to other participants (lines 18 and 20-24). By construction, all correct nodes in the sink component share the same partial view of the system (exactly the members in the sink – Lemma 1). Thus, these nodes know at least $2f+1$ correct members that belong to the sink component, what makes possible to reach the properties of the classical Byzantine consensus (Section 2.3);
2. Other nodes (in the remaining components) do not participate to the classical consensus. These nodes ask for the decison value to all known nodes, i.e., all reachable nodes, what includes all nodes in the sink (line 15). Each node 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 25-31). Theorem 1 shows that Algorithm 4 solves the BFT-CUP problem as defined in Section 2.3 with the stated participant detector and connectivity requirements.

Algorithm 4. Algorithm CONSENSUS executed at 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 that have required the decision value
 7. i.values : set of ⟨node, value⟩ tuples     // reported decisions

message:
 8. SET_DECISION:                              // struct of the message SET_DECISION
 9.     decision : value                       // the decided value

** All Nodes **
INIT: {Main Decision Task}
10. i.decision ←⊥; i.values ← ∅; i.asked ← ∅;
11. (i.in_the_sink, i.known) ← SINK();
12. if i.in_the_sink then
13.     Consensus.propose(i.initial);          // underlying Byzantine consensus with all
        p ∈ i.known
14. else
15.     reachable_send(GET_DECISION, i);
16. end if

** Node In Sink **
upon Consensus.decide(v)
17. i.decision ← v;
18. ∀j ∈ i.asked, send SET_DECISION(i.decision) to j;
19. return i.decision;

upon execution of reachable_deliver(GET_DECISION, sender)
20. if i.decision = ⊥ then
21.     i.asked ← i.asked ∪ {sender};
22. else
23.     send SET_DECISION(i.decision) to sender;
24. end if

** Node Not In Sink **
upon receipt of SET_DECISION(m.decision) from sender
25. if i.decision = ⊥ then
26.     i.values ← i.values ∪ {⟨sender, m.decision⟩};
27.     if #⟨∗,m.decision⟩ i.values ≥ f + 1 then
28.         i.decision ← m.decision;
29.         return i.decision;
30.     end if
31. end if
```

Theorem 1. *Consider a classical Byzantine consensus protocol. Algorithm CONSENSUS solves BFT-CUP, in spite of $f < \frac{k}{2} < n$ failures, if k-OSR PD is used and assuming at least $3f+1$ participants in the sink.*

Sketch of Proof. In this proof we have to consider two cases:
Processes in the sink: All correct participants in the sink component determine that they belong to the sink (Lemma 2) (line 12) and start the execution of an underlying classical Byzantine consensus algorithm (line 13). Then, as the sink has at least $2f+1$ correct nodes, it is guaranteed that all properties of the classical consensus will be met, i.e., *validity*, *integrity*, *agreement* and *termination*. Thus, nodes in the sink obtain the decision value (line 17), send this value to other participants (line 18) and return the decided value to the application (line 19), ensuring *termination*. Whenever a process in the sink receives a request for decision from other processes (lines 20–24), it will send the value if it has already decided (line 23); otherwise, it will store the sender's identity in order to send the decision value later (line 18) after the consensus has been achieved.

Processes not in the sink: Processes not in the sink request the decision value to all participants in the sink (line 15). Notice that if there is enough connectivity ($k \geq 2f+1$), nodes in the sink are reachable from any node of the system. Moreover, by properties of the reachable reliable broadcast, all correct participant in the sink will receive requests sent by correct participants not in the sink, even in the presence of $f < \frac{k}{2}$ failures (lines 20–24). Thus, as there are at least $2f+1$ correct participants in the sink able to send back replies for these requests (lines 18, 23), it is guaranteed that nodes not in the sink will receive at least $f+1$ messages with the same decision value (lines 25-31) and the predicate of line 27 will be true, allowing the process to *terminate* and return the decided value (line 28). Moreover, a collusion of up to f malicious participants cannot lead a process to decide for incorrect values (line 27), guaranteeing thus *agreement*. *Integrity* is ensured through the verification of predicate on line 25, by which each correct participant decides only once. Notice that *validity* is ensured through the underlying classical Byzantine consensus protocol, i.e., the decided value is a value proposed by nodes in the sink. This proves that *k*-OSR PD is sufficient to solve BFT-CUP. □

4.4 Necessity of *k*-OSR Participant Detector to Solve BFT-CUP

Using a *k*-OSR PD, our protocol requires a degree of connectivity $k \geq 2f+1$ to solve BFT-CUP. Theorem 2 states that a participant detector of this class and this connectivity degree are necessary to solve BFT-CUP.

Theorem 2. *A participant detector $PD \in$ k-OSR is necessary to solve BFT-CUP, in spite of $f < \frac{k}{2} < n$ failures.*

Sketch of Proof. This proof is based on the same arguments to prove the necessity of OSR (One Sink Reducibility) for solving CUP [6]. Assume by contradiction that there is an algorithm which solves BFT-CUP with a *PD* $\notin$ *k-OSR*. Let G_{di} be the knowledge graph induced by *PD*, then two scenarios are possible: (*i.*) there are less than *k* node-disjoint paths connecting a participant *p* in G_{di}; or (*ii.*) the directed acyclic graph

obtained by reduction of G_{di} to its k-strongly connected components has at least two sinks. There are two possible scenarios to be considered.

In the first scenario, let at most $2f$ node-disjoint paths connect p in G_{di}. Then, the simple crash failure of f neighbors of p makes it impossible for a participant i (being p reachable from i) to discover p, because only f processes are able to inform i about the presence of p in the system. In fact, i is not able to determine if p really exists, i.e., it is not guaranteed that i has received this information from a correct process. Then, the partial view obtained by i will be inconsistent, what makes it impossible to solve BFT-CUP. Thus, we reach a contradiction.

In the second scenario, let G_1 and G_2 be two of the sink components and consider that participants in G_1 have proposition value v and participants in G_2 value w, with $v \neq w$. By *Termination* property of consensus, processes in G_1 and G_2 must eventually decide. Let us assume that the first process in G_1 that decides, say p, does so at time t_1, and the first process in G_2 that decides, say q, does so at time t_2. Delay all messages sent to G_1 and G_2 such that they are received after $max\{t_1, t_2\}$. Since the processes in a sink component are unaware of the existence of other participants, p decides v and q decides w, violating the *Agreement* property of consensus and reaching thus a contradiction. □

5 Discussion

This section presents some comments about the protocol presented in this paper.

5.1 Digital Signatures

It is worth to notice that the lower bound required to solve BFT-CUP in terms of connectivity and resiliency is $k \geq 2f+1$, and it holds even if digital signatures are used. By using digital signatures, it is possible to exchange messages among participants, since there is at least one path formed only by correct processes ($k \geq f+1$). However, even with digital signatures, a connectivity of $k \geq 2f+1$ is still required in order to discover the participants properly (first phase of the protocol). In fact, if $k < 2f+1$, a malicious participant can lead a correct participant p not to discover every node reachable from it, what makes it impossible to use this protocol to solve BFT-CUP (the partial view of p will be inconsistent).

For example, Figure 2 presents a knowledge connectivity graph induced by a 2-OSR PD ($k=2$) in which the system does not support any fault (to support $f=1$, $k \geq 3$). Now, consider that process 2 is malicious and that process 1 is starting the DISCOVERY phase. Then, process 2 could inform to process 1 that it only knows process 3. At this point, process 1 will break the search because it is only waiting for a message from process 3, i.e., number of pending messages less or equal to f. Thus, process 1 obtains the wrong partial view $\{1,2,3\}$ of the system.

5.2 Protocol Limitations

The model used in this study, as well as in all solutions for FT-CUP [7,8], supports mobility of nodes, but it is not strong enough to tolerate arbitrary churn (arrivals and

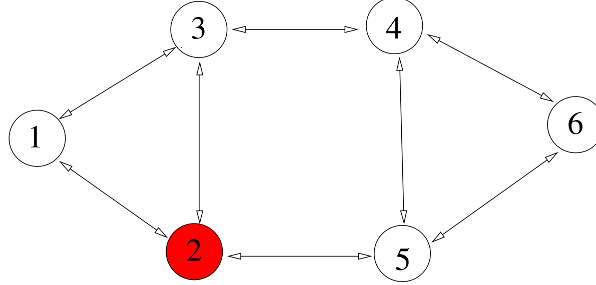


Fig. 2. 2-OSR with Process 2 Faulty

departures of processes) during protocol executions. This happens because, after the relations of knowledge have been established (first phase of the protocol), new participants will be considered only in future executions of consensus.

In current algorithms, process departures can be considered as failures. Nonetheless, this is not the optimal approach, since our protocols tolerate Byzantine faults and the behaviour of a departing process resembles a simple crash failure. An alternative approach consists in specifying an additional parameter d to indicate the number of supported departures, separating departures from malicious faults. In this way, the degree of connectivity in the knowledge graph should be $k \geq 2f + d + 1$ to support up to f malicious faults and up to d departures. Moreover, even with departures, the sink component should remains with enough participants to execute a classical consensus, i.e., $n_{sink} \geq 3f + 2d + 1$, following the same reasoning as [19].

5.3 Other Participant Detectors

Although k-OSR PD is the weakest participant detector defined to solve FT-CUP, there are other (stronger) participant detectors able to solve BFT-CUP [6,8]:

- FCO (Full Connectivity PD): the knowledge connectivity graph $G_{di} = (V, \xi)$ induced by the PD oracle is such that for all $p, q \in \Pi$, we have $(p, q) \in \xi$.
- k-SCO (k-Strong Connectivity PD): the knowledge connectivity graph $G_{di} = (V, \xi)$ induced by the PD oracle is k-strongly connected.

Notice that a characteristic common to all participant detectors able to solve BFT-CUP (except for the FCO PD that is fully connected) is the degree of connectivity k, which makes possible the proper work of the protocol even in the presence of failures. Using these participant detectors (FCO or k-SCO) the partial view obtained by each process in the system contains exactly all processes in the system (first phase of the protocol). Thereafter, the consensus problem is trivially solved using a classical Byzantine consensus protocol, since all processes have the same (complete) view of the system.

6 Final Remarks

Most of the studies about consensus found in the literature consider a static known set of participants in the system (e.g., [1,3,4,5,17,19]). Recently, some works which

Table 1. Comparing solutions for the consensus with unknown participants problem

Approach	failure model	participant detector	k	participants in the sink	connectivity between components	synchrony model
CUP [6]	without failures	OSR	–	1	OSR	asynchronous
FT-CUP [7]	crash	OSR	–	1	OSR + safe crash pattern	asynchronous + $\mathscr{P}$
FT-CUP [8]	crash	k-OSR	$f+1$	$2f+1$	k node-disjoint paths	asynchronous + $\diamond\mathscr{S}$
BFT-CUP (this paper)	Byzantine	k-OSR	$2f+1$	$3f+1$	k node-disjoint paths	same of the underlying consensus protocol

deal with a partial knowledge about the system composition have been proposed. The works of [6,7,8] are worth noticing. They propose solutions and study conditions in order to solve consensus whenever the set of participants is unknown and the system is asynchronous. The work presented herein extends these previous results and presents an algorithm for solving FT-CUP in a system prone to Byzantine failures. It shows that to solve Byzantine FT-CUP in an environment with little synchrony requirements, it is necessary to enrich the system with a greater degree of knowledge connectivity among its participants. The main result of the work is to show that it is possible to solve Byzantine FT-CUP with the same class of participant detectors (k-OSR) and the same synchrony requirements ($\diamond\mathscr{S}$) necessary to solve FT-CUP in a system prone to crash failures [8]. As a side effect, a Byzantine fault-tolerant dissemination primitive, namely *reachable reliable broadcast*, has been defined and implemented and can be used in other protocols for unknown networks.

Table 1 summarizes and presents a comparison with the known results regarding the consensus solvability with unknown participants.

Acknowledgements

Eduardo Alchieri is supported by a CAPES/Brazil grant. Joni Fraga and Fabíola Greve are supported by CNPq/Brazil grants. This work was partially supported by the EC, through projects IST-2004-27513 (CRUTIAL), by the FCT, through the Multiannual (LaSIGE) and the CMU-Portugal Programmes, and by CAPES/GRICES (project TISD).

References

1. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programing Languages and Systems 4(3), 382–401 (1982)
2. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
3. Toueg, S.: Randomized Byzantine Agreements. In: Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, pp. 163–178 (1984)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)

5. Correia, M., Neves, N.F., Veríssimo, P.: From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. The Computer Journal 49(1) (2006)
6. Cavin, D., Sasson, Y., Schiper, A.: Consensus with unknown participants or fundamental self-organization. In: Nikolaidis, I., Barbeau, M., Kranakis, E. (eds.) ADHOC-NOW 2004. LNCS, vol. 3158, pp. 135–148. Springer, Heidelberg (2004)
7. Cavin, D., Sasson, Y., Schiper, A.: Reaching agreement with unknown participants in mobile self-organized networks in spite of process crashes. Technical Report IC/2005/026, EPFL - LSR (2005)
8. Greve, F.G.P., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: Proceedings of the International Conference on Dependable Systems and Networks - DSN, pp. 82–91 (2007)
9. Castro, M., Liskov, B.: Practical Byzantine fault-tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
10. Douceur, J.: The sybil attack. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems (2002)
11. Awerbuch, B., Holmer, D., Nita-Rotaru, C., Rubens, H.: An on-demand secure routing protocol resilient to byzantine failures. In: Proceedings of the 1st ACM workshop on Wireless security - WiSE, pp. 21–30. ACM, New York (2002)
12. Kotzanikolaou, P., Mavropodi, R., Douligeris, C.: Secure multipath routing for mobile ad hoc networks. In: Wireless On-demand Network Systems and Services - WONS, pp. 89–96 (2005)
13. Papadimitratos, P., Haas, Z.: Secure routing for mobile ad hoc networks. In: Proceedings of SCS Communication Networks and Distributed Systems Modeling and Simulation Conference - CNDS (2002)
14. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of ACM 35(2), 288–322 (1988)
15. Bracha, G.: An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In: Proceedings of the 3rd ACM symposium on Principles of Distributed Computing, pp. 154–162 (1984)
16. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: Proceedings of the 2rd Annual ACM Symposium on Principles of Distributed Computing, pp. 27–30 (1983)
17. Friedman, R., Mostefaoui, A., Raynal, M.: Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. IEEE Transactions on Dependable and Secure Computing 2(1), 46–56 (2005)
18. Dolev, D.: The Byzantine generals strike again. Journal of Algorithms (3), 14–30 (1982)
19. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Transactions on Dependable and Secure Computing 3(3), 202–215 (2006)