Consensus with Unknown Participants or Fundamental Self-Organization*

David Cavin, Yoav Sasson, and André Schiper

École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Switzerland
{david.cavin, yoav.sasson, andre.schiper}@epfl.ch

Abstract. We consider the problem of bootstrapping self-organized mobile ad hoc networks (MANET), i.e. reliably determining in a distributed and self-organized manner the services to be offered by each node when neither the identity nor the number of the nodes in the network is initially available. To this means we define a variant of the traditional consensus problem, by relaxing the requirement for the set of participating processes to be known by all at the beginning of the computation. This assumption captures the nature of self-organized networks, where there is no central authority that initializes each process with some context information. We consider asynchronous networks with reliable communication channels and no process crashes and provide necessary and sufficient conditions under which the problem admits a solution. These conditions are routing and mobility independent. Our results are relevant for agreement-related problems in general within self-organized networks.

1 Introduction

The paper addresses the problem of bootstrapping a self-organized MANET. More precisely, the paper addresses the following question. Consider some geographical region R that is initially empty. At some point, one or more mobile nodes enter the region and want to deploy one or more services. However, to deploy the service(s), it is necessary for the first nodes that enter R to agree on an initial set of nodes, in order for these nodes to decide which node is going to provide what service. Let us call these nodes *I-nodes* (Infrastructure nodes).

To decide which node is going to provide which service, we need to solve an agreement problem that decides on a set *I-nodes*, and outputs *I-nodes* at each node in *I-nodes*. Once this problem is solved, each node in *I-nodes*, based on the knowledge of *I-nodes*, can locally determine which node is responsible for providing what service(s). For example, assume that the agreement is on *I-nodes* $= \{n_1, n_2, n_3\}$, and consider that there are five services s_1 to s_5 to provide. Based on the knowledge of *I-nodes*, n_1 will provide the services s_1 and s_2, n_2 will provide the services s_3 and s_4, and n_3 will provide the service s_5. Or, if there is

* The work presented in this paper was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant 5005-67322.

only one service to provide, n_1 will provide it, and n_2, n_3 know that they have no service to provide.

The problem is easily solved if there is some fixed node fn that is always in R: fn can act as a centralized decision point. However, this solution is not self-organized, since it relies on some preexisting infrastructure. This leads to the following question: is it possible to solve the problem without any preexisting centralized infrastructure, i.e., in a fully self-organized way?

Deciding on the set *I-nodes* can be modeled as a consensus problem [1]. In the consensus problem, a set Π of processes have to agree on a common value (called the decision value) that is the initial value of one of the processes. Consensus has been extensively studied in traditional networks with process failures, and algorithms based on various system models have been developed [2–4]. The importance of consensus is due to the fact that it is a basic building block for solving several other important fault-tolerant distributed problems. However, there is a fundamental difference between the classical consensus problem, and the problem addressed in the paper: in the paper *the set Π is unknown* (and Π is precisely the information we want to obtain). This makes it a new problem that we call *Consensus with Unknown Participants* or simply *CUP*. Note that the notion of consensus with *uncertain* participants appears in [5]. However, the specification is different, and the context is also different (it is used as a building block for implementing a dynamic atomic broadcast service in a wired synchronous network). Thus, the results in [5] are unrelated to the results established in this paper.

The CUP problem is formally defined in Section 2, which also defines the model in which the problem is solved. The classical consensus problem is hard to solve because processes may crash. With CUP, the difficulty of the problem is due to the *unknown* participants. So, for simplification, we assume in the paper that processes (i.e., mobile nodes) do not crash. We also assume that the nodes in R always form a connected network, and we assume the existence of an underlying multihop routing protocol: if some node n knows the existence of a node n' in R, then n can reliably send a message to n'. *Moreover, n can only send reliably a message to nodes that it knows.* Given these assumptions, the results obtained in the paper are independent of the underlying routing algorithm or mobility pattern of the nodes.

Clearly if all nodes in R do not know any other nodes, then no node can send any message to any other node, and CUP cannot be solved. This leads us to add to our model the notion of *participant detectors*, which are distributed oracles attached to each node n. The participant detector of n provides to n a (possibly small) subset of the nodes in R, e.g. by receiving messages or beacons. In Section 3 we introduce various classes of participant detectors, and we compare them based on the classical notion of reduction. In Section 4 we identify necessary and sufficient conditions for solving CUP, and we solve CUP using the participant detector named *one sink reducibility.* In Section 5 we illustrate how CUP can be used for solving the bootstrapping problem described in this section. Finally, we conclude and present future work in Section 6.

2 Consensus with Unknown Participants

We consider a finite set Π of processes drawn from a (finite or infinite) universe $\mathcal{U}$. The processes in Π have to solve the traditional consensus problem, but contrary to the usual model for consensus, the processes in Π do not necessarily know each other. This assumption captures the *self-organization* nature of the type of system that we consider: there is no central authority that initializes each process with some context information.

To solve consensus, processes in Π communicate by message passing. However, *process $p \in \mathcal{U}$ can send a message to process $q \in \mathcal{U}$ iff p knows the existence of q.* Similarly, process q can send a message to p iff q knows the existence of p. So if p knows q, but q does not know p, the communication is asymmetric. If q does not know p and receives a message from p, from there on q knows p, i.e., q can send a message to p. Communication channels are reliable and the system is asynchronous: we do not assume any bound on the transmission delay of messages nor on the process relative speeds. We finally make the strong assumption that processes never crash. Indeed, the necessary conditions established in the paper are still true in models with process crashes.

The specification of consensus with unknown participants (CUP) is close to the classical specification of consensus [6]: an instance of the CUP problem is defined by the primitives $propose(v_i)$ by which each process $p_i \in \Pi$ proposes its initial value v_i, and $decide(v)$, by which a process decides on a value v. The decision must satisfy the following conditions:

Validity If a process decides v, then v is the initial value of some process.
Agreement Two processes cannot decide differently.
Termination Every process eventually decides.

In the classical consensus problem processes know each other, i.e., processes in Π know Π. This is not the case here. Moreover, in the classical consensus problem processes may crash. The crash of processes makes the problem difficult. Here processes do not crash; what makes the problem difficult is the ignorance of the set of other processes having to solve an instance of the consensus problem.

3 Participant Detectors

3.1 Presentation and Specification

If each process $p \in \Pi$ knows only itself, then p cannot communicate with any other process in Π, which clearly makes it impossible to solve consensus. We are interested in identifying the minimal information that processes must have about the other participants to make consensus solvable. We capture the information that process p has about other processes by the notion of *participant detectors*. Similarly to failure detectors [4], participant detectors are distributed oracles associated with each process. In the setting of the classical consensus problem, where the set of participants is *known*, the task of failure detectors is to maintain a set of processes which are suspected to have crashed. In our context however, the set of participants is *unknown.* By querying their local participant detector, processes may obtain an approximation of Π, the set of processes participating in consensus.

We denote by PD_p the participant detector of process p. Process p can query its participant detector PD_p, which returns a set of processes. We denote by $PD_p(t)$ the query of p at time t. The information returned by PD_p can evolve between queries, but verifies the following two properties.

Property 1 (Information Inclusion). The information returned by the participant detectors is non-decreasing over time:

$$\forall p \in \Pi, \forall t' \geq t : \ PD_p(t) \subseteq PD_p(t')$$

Property 2 (Information Accuracy). The participant detectors do not make mistakes in the sense that they do not return a process that does not belong to Π:

$$\forall p \in \Pi, \forall t : \ PD_p(t) \subseteq \Pi$$

PD_p gives p an initial context, i.e., an approximation of Π. This context will allow p to start sending messages to other processes. Moreover, messages received by p will allow p to increase its knowledge of Π. We define below participant detectors reflecting different levels of accuracy of participant estimation. To define these detectors, we consider

1. The (undirected) graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the vertices $\mathcal{V} = \Pi$ and the (undirected) edge $(p, q) \in \mathcal{E}$ iff $q \in PD_p$ or $p \in PD_q$.
2. The directed graph $\mathcal{G}_{di} = (\mathcal{V}, \mathcal{E})$, where the vertices $\mathcal{V} = \Pi$ and the directed edge $(p, q) \in \mathcal{E}$ iff $q \in PD_p$.

Participant Detector 1 (Connectivity CO). *A participant detector satisfies the* connectivity *property iff the (undirected) graph $\mathcal{G}$ is connected.*

Participant Detector 2 (Strong Connectivity SCO). *A participant detector satisfies the* strong connectivity *property iff the directed graph $\mathcal{G}_{di}$ is strongly connected.*

Participant Detector 3 (Full Connectivity FCO). *A participant detector satisfies the* full connectivity *property iff the directed graph $\mathcal{G}_{di} = (\Pi, \mathcal{E})$ is such that for all $p, q \in \Pi$, we have $(p, q) \in \mathcal{E}$.*

Finally, we introduce the last detector, satisfying the *one sink reducibility* property. The motivation behind this particular participant detector will become clear to the reader in Section 4.

Participant Detector 4 (One Sink Reducibility OSR). *A participant detector satisfies the* one sink reducibility *property iff the graph $\mathcal{G}$ is connected* *and the directed acyclic graph obtained by reducing $\mathcal{G}_{di}$ to its strongly connected components has one and only one sink*[1].

Note that Properties 1 and 2 allow processes to query their participant detectors at different times. It is easy to see that if some participant detector PD satisfies the the property of CO, SCO, FCO or OSR if queried at some time t, it also satisfies the property when queried at some time $t' > t$. Consider for example OSR and assume that PD satisfies OSR at time t. If the graph $\mathcal{G}_{di}$ reduced to its strongly connected components contains at most one sink, adding edges to $\mathcal{G}_{di}$ at time t' cannot increase the number of sinks.

3.2 Comparing Participant Detectors

Given the system model of Section 2, we are interested in establishing relationships between participant detectors. To this means we use the notion of *reducibility*, borrowed from the concepts introduced in [4] for comparing failure detectors.

Definition 1 (Reducibility). *A participant detector PD is* reducible *to a participant detector PD', noted $PD \succeq PD'$, if there exists an algorithm $A_{PD \to PD'}$ that transforms the participant detector PD into the participant detector PD'.*

In other words, if a participant detector PD is reducible to PD', it is possible to use PD in order to emulate PD'. PD' is said to be weaker than PD: any problem that can be solved using PD' can be solved using PD instead. Figure 1 illustrates the reduction.

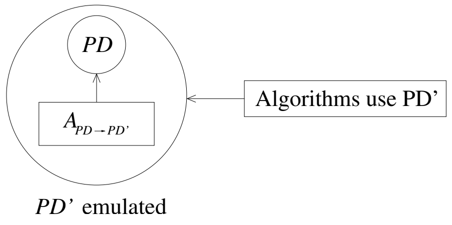

Fig. 1. Transforming Participant Detectors

Definition 2 (Equivalence). *If PD is reducible to PD' ($PD \succeq PD'$) and PD' is reducible to PD ($PD' \succeq PD$), then we say that PD and PD' are* equivalent*, noted by $PD \cong PD'$.*

[1] A *sink* in a directed graph is a vertex with out-degree 0, i.e. there are no edges leaving the vertex. An example graph representing $PD \in OSR$ is provided in Figure 2, which depicts a graph with four strongly connected components A, B, C, D and one sink, namely C.

If two participant detectors are equivalent, they belong to the same *equivalence class*. Any result that applies to one (e.g. problem that can be solved, impossibility result), applies to the second as well.

4 Solving CUP

4.1 Preliminary Results

The goal of this section is to identify the necessary and sufficient requirements for solving CUP by exploring the participant detectors presented in Section 3. We begin with the following intuitive elementary result.

Proposition 1. *The Connectivity participant detector is necessary but not sufficient to solve CUP.*

Proof.
i) Necessary: Assume that the resulting graph $\mathcal{G}$ returned by the participant detectors is disconnected, i.e. there exists two components C_1 and C_2 of processes that cannot communicate with each other and independently execute consensus. Let v_1 be the initial value of the processes in C_1 and v_2 of those in C_2 (with $v_2 \neq v_1$). Consensus terminates in both components, with processes in C_1 deciding on v_1 and processes in C_2 on v_2, leading to a violation of the agreement property.
ii) Not sufficient: Consider $\Pi = \{p_1, p_2, p_3\}$ such that $PD_{p_1} = \{p_1, p_2, p_3\}$, $PD_{p_2} = \{p_2\}$ and $PD_{p_3} = \{p_3\}$. Let v_2 be the initial value of p_2, v_3 the initial value of p_3, with $v_2 \neq v_3$. Clearly, $PD \in CO$. Processes p_2 and p_3, unaware of any other process besides themselves, execute consensus and decide on values v_2 and v_3 respectively, violating the agreement property. $\square$

The Full Connectivity Oracle is a trivial oracle in the sense that it fully compensates for the uncertainty of the participating processes, leading to the following proposition:

Proposition 2. *The Full Connectivity participant detector is sufficient but not necessary to solve CUP.*

Proof.
Sufficient: It is easy to see that Algorithm 1 (on page 7) meets the consensus specification given our model. Indeed, the Full Connectivity participant detector provides the full set Π of participants to all processes. Therefore, each process can deterministically choose a common leader, e.g. the process with the lowest identifier. The leader process sends its initial value v to all processes in Π, which decide on v upon reception.
Not necessary: Consider $\Pi = \{p_1, p_2, p_3\}$, with $PD_{p_1} = \{p_1, p_2, p_3\}$, $PD_{p_2} = \{p_1, p_2\}$ and $PD_{p_3} = \{p_2, p_3\}$. We have that $PD \notin FCO$. Nevertheless, the following algorithm solves consensus. Let p_i^{min} denote the smallest process returned by the participant detector of p_i. If $p_i = p_i^{min}$ then p_i decides on its own initial value and sends the value to PD_{p_i}. If $p_i \neq p_i^{min}$, then p_i waits for a value and decides upon reception. $\square$

Algorithm 1 Solving consensus with $PD \in FCO$ for a process $p_i \in \Pi$

```
1: propose($v_i$):
2: $participants_i \leftarrow PD_i$;
3: $leader_i \leftarrow min(participants_i)$;
4: if $p_i = leader_i$ then
5:    $decision_i \leftarrow v_i$;
6:    send $decision_i$ to all $p_j \in participants_i$;
7: end if
8:
9: {Upon receive($decision$):}
10: decide($decision$);
```

We now establish the equivalence between the Strong Connectivity and Full Connectivity participant detectors.

Proposition 3. *The Strong Connectivity and Full Connectivity participant detectors are equivalent.*

Proof. With a Full Connectivity participant detector, each process can communicate with every other process in Π, hence $\mathcal{G}_{di}$ is strongly connected and $PD \in FCO$ trivially implies $PD \in SCO$, i.e., $FCO \succeq SCO$.

We prove $SCO \succeq FCO$ with Algorithm 2, a token-based depth-search algorithm that discovers all members of Π assuming $PD \in SCO$. Every process $p_i \in \Pi$ initiates the participant discovery by invoking `discover_participants()` after having queried its participant detector $PD \in SCO$ (l.4). Notice that processes query their participant detector only once. Every token carries the following information :

$token_i.issuer$ **:** The identity of the process having issued the token.
$token_i.visited$ **:** A set containing the processes visited by the token.
$token_i.tovisit$ **:** A set containing the known processes that have not yet been visited by the token.

Prior to forwarding the token, p_i adds in $token_i.tovisit$ the processes it can communicate with, i.e. processes returned by its local strong connectivity participant detector (l.8). The token is then forwarded to any process present in $token_i.tovisit$ (l.9). Upon reception of $token_i$ by a process p_j, p_j checks whether it is the issuer of $token_i$. If yes, the algorithm terminates and returns the set of visited processes (l.13). If not, it updates the data structures stored in $token_i$ as follows. First, it adds to $token_i.tovisit$ all processes in PD_{p_j} that have not yet been visited (l.15). p_j then removes its own ID from $token_i.tovisit$ (l.16), adds it in $token_i.visited$ (l.17). If there are no more processes to visit, p_j sends the token back to $token_i.issuer$ (l.18-19). Otherwise, it simply forwards $token_i$ to any one of them (l.21).

In order to prove the correctness of the algorithm, we need to show that it terminates and returns all processes. Consider the token issued by p_i. To prove that the algorithm terminates, we must show that p_i eventually executes line 13. This happens when $token_i$ (the token issued by p_i) has returned to p_i. Assume that $token_i$ is located at some process p_j. Either $token_i.tovisit$ contains some process not visited by the token, in which case p_j forwards the token to one of them or p_j sends back $token_i$ to $token_i.issuer$. Since the number of processes is finite, eventually all processes are visited, in which case line 19 is executed and eventually p_i executes line 13.

It remains to prove that when process p_i executes line 13, $token_i.visited$ contains all the processes. We prove the result by contradiction. Assume that $token_i$ is located at some process p_j, and p_j sends back the token by executing line 19 while $token_i.visited$ does not contain some processes X. This means that for all processes $p_j \in token_i.visited$, PD_{p_j} does not contain any of the processes in the set X. A contradiction with SCO. □

Algorithm 2 Participant discovery algorithm for process $p_i \in \Pi$

```
 1: $token_i.issuer \leftarrow p_i$;
 2: $token_i.visited \leftarrow \emptyset$;
 3: $token_i.tovisit \leftarrow \emptyset$;
 4: $neighbors_i \leftarrow PD_i$; {Participant detector invocation}
 5:
 6: discover_participants():
 7: $token_i.visited \leftarrow \{p_i\}$;
 8: $token_i.tovisit \leftarrow neighbors_i \setminus \{p_i\}$;
 9: send $token_i$ to any $p_j \in token_i.tovisit$;
10:
11: {Upon receive($token_j$) from $p_k$ :}
12: if $token_j.issuer = p_i$ then
13:   return $token_j.visited$; {Algorithm terminates}
14: else
15:   $token_j.tovisit \leftarrow token_j.tovisit \cup (neighbors_i \setminus token_j.visited)$;
16:   $token_j.tovisit \leftarrow token_j.tovisit \setminus \{p_i\}$;
17:   $token_j.visited \leftarrow token_j.visited \cup \{p_i\}$;
18:   if $token_j.tovisit = \emptyset$ then
19:     send $token_j$ to $token_j.issuer$;
20:   else
21:     send $token_j$ to any $p_l \in token_j.tovisit$;
22:   end if
23: end if
```

Since the FCO and SCO participant detectors are equivalent, it is straightforward to obtain an algorithm that solves CUP with SCO: execute Algorithm 1 with FCO, where FCO is obtained from SCO using Algorithm 2.

It follows from Proposition 2 and Proposition 3 that the Strong Connectivity detector is also sufficient but not necessary for solving CUP. Since by Proposition 1, CO is necessary but not sufficient, we must look for a participant detector somewhere in between CO and SCO for it to be both necessary and sufficient for solving CUP. The OSR participant detector, introduced in the next section, is such a participant detector.

4.2 OSR is necessary and sufficient

The following proposition is the main result of the paper and serves as a basis for designing algorithms that solve CUP.

Proposition 4. *The One Sink Reducibility participant detector is necessary and sufficient to solve CUP.*

Proof.
i) Necessary: Suppose that $PD \notin OSR$, i.e. the directed acyclic graph obtained by reduction of $\mathcal{G}_{di}$ to its strongly connected components has at least two sinks S_1 and S_2. Since there is no outgoing path leaving components S_1 and S_2, processes in S_1 and S_2 can be unaware of the existence of the other sink. We prove the result by contradiction.

Assume there exists an algorithm $\mathcal{A}$ that solves CUP. Let all initial values of processes in S_1 be different from all initial values of processes in S_2. By the termination property of consensus, processes in S_1 and processes in S_2 must eventually decide. Let us assume that the first process in S_1 that decides, say p, does so at t_1, and the first process in S_2 that decides, say q, does so at t_2. Delay all messages sent to S_1 and S_2 such that they are received after $max(t_1, t_2)$. So the decision of p is on the initial value of some process in S_1, and the decision of q is on the initial value of some process in S_2. Since these initial values are different, the agreement property of consensus is violated. A contradiction with the assumption that $\mathcal{A}$ solves CUP.

ii) Sufficient: The proof is by giving Algorithm 3 (see page 14) that solves CUP with $PD \in OSR$ (see next section). □

4.3 Solving CUP with OSR

Intuition of the algorithm

Algorithm 3 (on Page 14) solves CUP with $PD \in OSR$. We present the general intuition with help of the example graph of Figure 2.

The key strategy is to ensure that the nodes belonging to the sink of the graph obtained by reduction to its strongly connected components (component C in the figure) decide before other strongly connected components. The decision can then be appropriately propagated to the rest of the participants. Processes query their $PD \in OSR$ participant detector *only once*, at the beginning of the consensus execution. Initially, processes have no other knowledge besides that returned by the participant detector and are unable to discern whether or not they belong to the sink. To augment their knowledge, processes execute

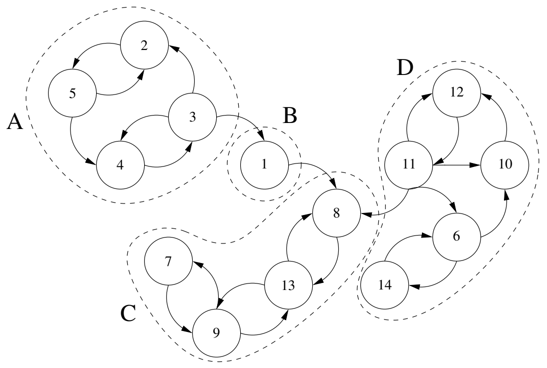

Fig. 2. An example graph $\mathcal{G}$ representing $PD \in OSR$, along with its strongly connected components (arrows represent the information provided by the participant detectors).

the token discovery algorithm presented in Algorithm 2. Although participant detectors are queried only once, a process can still discover and communicate with nodes discovered later in the course of the computation, e.g. by receiving a message from a process not returned by its PD. In Figure 2, processes in the strongly connected component A will, by executing the token discovery algorithm of Figure 2, discover processes in A, B and C. Similarly, processes in B will discover those in B and C; processes in D will discover those in D and C. Processes in C however will only discover the processes in the same component C. Let $discovered(p_i)$ denote the processes discovered by process p_i.

After the execution of Algorithm 2, every process p_i elects as a leader the process in $discovered(p_i)$ with the lowest identifier (line 11). In our example, process 1 will be the leader in components A and B, process 7 in C and process 6 in D. Non leader processes will send a *decisionRequest* message to their leader to get the decision value (line 17). The message is received at line 44; upon reception of this message the leader registers the decision request in the set $decisionRequestors_i$ (line 48).

The leaders then identify among themselves the leader of the sink component. In order to do so, each leader sends at line 15 the *am I the sink leader?* message to all the processes it has previously discovered (the set $participants_i$ in Algorithm 2), and waits for acknowledgments (*acks*) or negative acknowledgments (*nacks*) from *all* these processes. Upon receiving *am I the sink leader?* from a leader l, a process p either responds with *ack* if p's leader is l (line 24), or otherwise with $(nack, leader)$, where *leader* is p's leader (line 26).

The sink leader (process 7 in our example) will be the only leader to receive only *ack*s. Other leaders will receive one or more $(nack, leader_j)$ messages and will send a *decisionRequest* message to $leader_j$ to get the decision value (line 41). The message is received at line 44; upon reception of this message the leader either sends the decision if available (line 46) or registers the request using the set $decisionRequestors_i$ (line 48).

Finally, the sink leader sends its initial value to all the processes in the set $participants_i$ (line 33). The sink leader itself receives this message at line 51, decides at line 54 and then, using the $decisionRequestors_i$ set, sends the decision to all non sink leaders. The non sink leaders propagate the decision to the non leaders registered in their $decisionRequestors_i$ set. In our example, the sink leader (process 7) sends the decision to all processes in C, as well as to local leaders 1 and 6. Process 1 will propagate the decision to the processes in component A and process 6 to the processes in component D.

Note that the nodes in component C have been able to decide without information about participants outside of their component. The correctness of the algorithm is discussed below.

Correctness of Algorithm 3 (sketch)

We only give a sketch of the proof. We start with three lemmas.

Lemma 1. *If $PD \in OSR$, the set $participants_i$ returned by Algorithm 2 at line 10 contains (at least) every node of the sink component.*

Proof. Algorithm 2 ensures that the token created at a process p_i will visit every process reachable from p_i. Assuming $PD \in OSR$, there exists a path from p_i to every process in the sink, i.e., every process in the sink is reachable from p_i. □

Lemma 2. *If $PD \in OSR$, for any process in the sink component, the set returned by Algorithm 2 contains exactly every node of the sink component.*

Proof. By definition a sink is a strongly connected component without any outgoing link. So the discovery Algorithm 2 executed by a process in the sink component will return all processes in the sink component and no other. □

Lemma 3. *The leader of the sink component is the only process to impose its initial value as the decision.*

Proof. The leader of the sink component is the only leader that will receive only *ack*s, from every process of its *participants* set (l.30). So it will be the only process to send its proposal (l.33). □

With help of the above lemmas, we can show that Algorithm 3 meets the specifications of CUP (see Section 2).

- **Validity**. By line 32 the decision is the initial value of some process. So the validity property is trivially satisfied. □
- **Agreement**. By Lemma 3 the leader of the sink component is the only process to send its initial value as the decision. The agreement property trivially follows from this. □

Termination. To prove that Algorithm 3 terminates for every process, we must show that every process executes line 55 (after reception of the message $(decision_j, leader_j)$, line 51). We distinguish three cases :

- *Processes in the sink.* By Lemma 2 the set $participants_i$ of the sink leader contains all processes in the sink. So, by line 33 (sending of the decision by the sink leader to $participants_i$) all processes in the sink component eventually receive the decision message.
- *Local leaders.* Since there is only one leader (the sink leader) that receives only acks, all other leaders will receive nacks. Specifically, by Lemma 1 they will receive $(nack, leader)$ from the processes in the sink component, with $leader = sink\ leader$. So the local leaders will send $decisionRequest$ to the sink leader (l.41), and the sink leader will eventually send them the decision (l.55).
- *All other processes.* Every non leader process registers itself with its leader by sending the message $decisionRequest$ (l.17). Since every local leader eventually decides (previous case), every local leader will send the decision to all these processes (l.55). □

4.4 Necessary and Sufficient Participant Detectors for CUP

Figure 3 summarizes the relationships between the participant detectors and CUP.

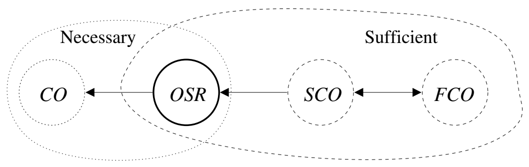

Fig. 3. Relationships between participant detectors

5 Bootstrapping a Self-Organized MANET

We now illustrate how CUP can be applied to solve the problem presented in the introduction, i.e. for bootstrapping a MANET in a self-organized manner. Let R be some geographical region that is initially empty. The mobile nodes have no prior knowledge of the identity nor of the number of peers in R (or in the entire network). At some point, one or more mobile nodes enter the region and want to deploy one or more services. However, to reliably deploy the service(s), it is necessary for the first nodes that enter R to agree on an initial set of infrastructure nodes, *I-nodes*, in order to determine which node is going to provide what service.

To obtain the set of *I-nodes*, each node in R executes the CUP algorithm described in Section 4.3, with the *participants* set of Algorithm 3 as the initial value. Since the decision is the initial value of the sink leader, the decision of CUP is the sink component. Let the set *I-nodes* be the decision of the CUP algorithm (i.e. the sink nodes). By the CUP specification (Section 2), this set will be the same for each node in R. Based on the knowledge of *I-nodes*, nodes in R can locally decide which node is responsible for providing what service(s). The unequivocal determination of the *I-nodes* through CUP leads to a unequivocal attribution of the services to be offered by each node.

6 Summary and Future Work

The paper has addressed the question of bootstrapping a self-organized MANET, i.e. for nodes that have no prior knowledge about their peers in the network to reliably agree on the set of services to be offered by each node. To this means we have introduced CUP, a variant of the consensus problem where neither the identity nor the number of participating processes is known. We have identified (mobility and routing independent) necessary and sufficient conditions, based on the notion of *participant detectors*, for which CUP admits a solution. We have also presented an algorithm, using the participant detectors called OSR, that enables nodes to reliably solve the aforementioned bootstrapping problem in a self-organized manner. Since consensus is a basic building block for solving other distributed algorithms, our results concerning CUP are in particular relevant for other agreement related problems (e.g. total-ordered broadcast, leader election) in self-organized settings such as MANET.

For future work, we intend to investigate the additional requirements necessary for solving CUP when processes may crash.

References

1. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. JACM **32** (1985) 374–382
2. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco, CS (1996)
3. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35** (1988) 288–323
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
5. Bar-Joseph, Z., Keidar, I., Lynch, N.: Early-delivery dynamic atomic broadcast. In Malkhi, D., ed.: Proceedings of the 16th International Symposium on Distributed Computing (DISC'02). Volume 2508 of Lecture Notes in Computer Science., Toulouse, France, Springer (2002) 1–16
6. Fischer, M.J.: The consensus problem in unreliable distributed systems (a brief survey). In: Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory, Springer-Verlag (1983) 127–140

Algorithm 3 Solving consensus with $PD \in OSR$ for a process $p_i \in \Pi$

```
1:
2:  $leader_i \leftarrow \perp$; {leader estimate}
3:  $leaders_i \leftarrow \emptyset$; {set of leaders}
4:  $decisionRequestors_i \leftarrow \emptyset$; {set of processes requiring a notification of the decision}
5:  $proposal_i \leftarrow false$; {initial value}
6:  $decision_i \leftarrow \perp$; {decision value}
7:  $participants_i \leftarrow \emptyset$; {set of processes returned by Algorithm 2}
8:
9:  **propose($v_i$):**
10: $participants_i \leftarrow discover\_participants()$; {execute and store result from Algorithm 2 using $PD \in OSR$}
11: $leader_i \leftarrow min(participants_i)$;
12: **if** $p_i = leader_i$ **then**
13:     {the process may be the sink leader, send a message}
14:     $proposal_i \leftarrow v_i$;
15:     send *am I the sink leader?* to all $p \in participants_i$;
16: **else**
17:     send *decisionRequest* to $leader_i$;
18: **end if**
19: {the procedure exits and the process waits for a decision}
20:
21: **Upon reception of** *am I the sink leader?* **from process** $p_j$**:**
22: **if** $p_j \neq leader_i$ **then**
23:     {disagreement on leader identity}
24:     send $(nack, leader_i)$ to $p_j$;
25: **else**
26:     send *ack* to $p_j$;
27: **end if**
28:
29: **Upon reception of** *ack* **from process** $p_j$**:**
30: **if** *ack* received from $\forall p \in participants_i$ **then**
31:     {the process is indeed the sink leader; propagate $proposal_i$ as the decision}
32:     $decision_i \leftarrow proposal_i$;
33:     send $(decision_i, leader_i)$ to all $p \in participants_i$;
34: **end if**
35:
36: **Upon reception of** $(nack, leader_j)$ **from process** $p_j$**:**
37: {the process is a local leader}
38: **if** $leader_j \notin leaders_i$ **then**
39:     {request a decision from $leader_j$}
40:     $leaders_i \leftarrow leaders_i \cup \{leader_j\}$;
41:     send *decisionRequest* to $leader_j$;
42: **end if**
43:
44: **Upon reception of** *decisionRequest* **from process** $p_j$**:**
45: **if** $decision_i \neq \perp$ **then**
46:     send $decision_i$ to $p_j$;
47: **else**
48:     $decisionRequestors_i \leftarrow decisionRequestors_i \cup \{p_j\}$;
49: **end if**
50:
51: **Upon reception of** $(decision_j, leader_j)$ **from process** $p_j$**:**
52: **if** $decision_i = \perp$ **then**
53:     $decision_i \leftarrow decision_j$;
54:     **decide($decision_j$)**;
55:     send $(decision_j, leader_j)$ to all $p \in decisionRequestors_i$;
56: **end if**
```