Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes

David Cavin Yoav Sasson André Schiper
{david.cavin,yoav.sasson,andre.schiper}@epfl.ch
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne

EPFL Technical Report IC/2005/026

Abstract

We consider the consensus problem in self-organized networks such as MANETs. Consensus offers a means for reliably solving agreement related problems, which in a self-organized setting can help bring enhanced structure and reliability to the highly dynamic and disorganized environment of MANETs. We consider asynchronous networks with reliable communications channels. Neither the identity nor the number of nodes is initially known to the participants. This captures the self-organized nature of the network, where no central authority initializes each process with some context information. In [CSS04], the authors have identified the conditions for which the above problem admits a solution when the processes are correct. The contribution of this paper is to determine the necessary and sufficient conditions for the above problem when processes can crash. These conditions are routing and mobility independent.

1 Introduction

We consider the consensus problem in the context of self-organized mobile networks such as MANETs. Consensus is a basic building block for solving important fault-tolerant distributed problems that require agreement among a set of processes: 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. In traditional wired networks, consensus is a cornerstone abstraction for software based replication techniques that offer fault tolerance.

MANETs, due to their highly distributed and self-organized nature can benefit immensely from consensus to achieve self-organization and reliability. For example, upon entering an empty geographic region, mobile nodes can reliably and unequivocally agree on which node shall deploy which service by executing consensus. Similarly to their wireline counterparts, a set of mobile nodes can offer fault-tolerant services by replicating it on a set of nodes by means of consensus, thus providing a reliable service above failure-prone mobile nodes.

Nevertheless, the traditional consensus specification is not transposable as such to MANETs due to their self-organized property. In particular, nodes in the network are not initialized with context information about the network infrastructure. Therefore, to the contrary of consensus in traditional networks, we cannot assume that processes in MANETs know which processes participate in the execution of consensus. Indeed, in a truly self-organized setting there is no central authority that can provide this information. To address this characteristic, [CSS04] specify a derived consensus problem adapted to the self-organized setting: Consensus with Unknown Participants or simply CUP. The authors in [CSS04] furthermore provide a solution to the problem in asynchronous networks when processes do not crash. This leads to the following question. Under what conditions, if at all, is it possible to solve CUP when processes can crash?

*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.
The contribution of our paper is twofold. Firstly, we extend the CUP specification and system model to incorporate process crashes, which we refer to by Fault-Tolerant Consensus with Unknown Participants (FT-CUP). Secondly, we identify the necessary and sufficient conditions for a solution to exist and provide an FT-CUP algorithm, thus offering a complete answer to consensus in MANETs by considering both the possibility for unknown participants and process crashes. Our solution assumes the existence of a reliable underlying multihop routing protocol: if some node \( n \) knows the existence of a node \( n' \), then \( n \) can reliably send a message to \( n' \). Given these assumptions, the results obtained in the paper are independent of the underlying routing algorithm or mobility pattern of the nodes.

In [BKP03], the authors survey dependability issues in mobile wireless networks. Due to the highly dynamic and ever-changing properties of MANET, the general approach in the literature is to construct more flexible algorithms that do not require strict compliance with the specification, but comply either eventually, with a satisfactory probability level, or simply on a best-effort basis. Our goal is to solve consensus reliably, i.e. without ever violating the specification. Note that the notion of consensus with uncertain participants appears in [BJKL02]. 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 [BJKL02] are unrelated to the results established in this paper.

The rest of the paper is organized as follows. We provide the FT-CUP specification in Section 2 as well as a description of the system model. In Section 3 we present the participant detector modules, which serve, when queried, to provide knowledge about the participating processes. Section 4 recalls the main results established in [CSS04] for the CUP problem in the particular setting where processes do not crash. We derive, in Section 5, the necessary and sufficient conditions for solving FT-CUP. Finally, we conclude in Section 6 by discussing our results and presenting future work.

## 2 Fault-Tolerant Consensus with Unknown Participants

We consider a finite set \( \Pi \) of processes drawn from a (finite or infinite) universe \( \mathcal{U} \). The processes in \( \Pi \) have to solve the traditional consensus problem, but contrary to the usual model for consensus, the processes in \( \Pi \) 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 \( \Pi \) 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 \( p \). In practice, this communication model assumes a reliable underlying routing algorithm. Hence, when a process \( p \) discovers the existence of some other correct process \( q \), it can, from that point on, rely on the routing algorithm to successfully deliver a message to \( q \). 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.

Self-organized networks such as MANETs evolve in a more dynamic and autonomous environment than their traditional wireline network counterparts. Therefore, in addition to straightforward crashes, processes in MANETs are even more prone to leave the network for reasons such as energy supply depletion, network disconnection due to limited transmission range or device power-off for energy saving. We address this behavior by considering a crash/no recovery model, i.e. processes in \( \Pi \) can leave the network, but they cannot recover in the sense that they will no longer be able to participate in same execution instance of consensus. We assimilate henceforth all faults to process crashes and employ the terms crashes and faults interchangeably in our discourse.

The consensus problem is defined by the primitives \( \text{propose}(v_i) \) by which process \( p_i \in \Pi \) proposes an initial value \( v_i \), and \( \text{decide}(v) \) by which a process decides on a value. The consensus decision satisfies the following validity, agreement and termination properties:
Validity If a process decides \( v \), then \( v \) is the initial value of some process (i.e., \( v \) was proposed by some process).

Agreement Two correct processes cannot decide differently.

Termination Every correct process eventually decides.

Note that the requirement for correct processes in the agreement property specifies non-uniform consensus, as we show necessary in our model by Proposition 5.1 of Section 5.1. The requirement for correct processes in the termination property defines consensus when processes can crash, augmenting the crash-free specification considered in [CSS04].

In summary, to better describe the reality of self-organized networks, our model introduces two key difficulties: to the contrary of classical consensus, the processes in \( \Pi \) do not know \( \Pi \). Furthermore, to the contrary of [CSS04], processes can crash.

### 3 Participant Detectors

If each process \( p \in \Pi \) knows only itself, then \( p \) cannot communicate with any other process in \( \Pi \), which clearly makes it impossible to solve consensus. We capture the information that process \( p \) has about other processes by the notion of participant detectors introduced in [CSS04]. Similarly to failure detectors [CT96], participant detectors are distributed oracles associated with each process. In the setting of the classical consensus problem the set of participants is known. In our context however, the set of participants is unknown. By querying their local participant detector, processes can obtain an approximation of \( \Pi \), 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 (the motivation behind these properties is given at the end of the section).

**Property 3.1 (Information Accuracy).** The participant detectors do not make mistakes in the sense that they do not return a process that does not belong to \( \Pi \):

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

**Property 3.2 (Information Inclusion).** The information returned by the participant detectors is non-decreasing over time:

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

The Information Accuracy property can be implemented by having the processes exchange beacons that contain identity information (we do not consider the case of malicious processes). The Information Inclusion property is trivially satisfied by never discarding the identity of detected processes.

Participant detectors can be defined to reflect different levels of accuracy of participant estimation. To define these detectors, we consider

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

Intuitively, network connectivity is a necessary condition for consensus. Indeed, by the termination property of the consensus specification every correct process must decide, even if disconnected. A disconnected process deciding on its own can lead to a violation of the agreement property. At the other end of the spectrum, if the participant detector is guaranteed to provide the full set of processes \( \Pi \), then the FT-CUP problem is no different than traditional consensus in wired networks. The following \( CO \) and \( FCO \) participant detectors specify these two boundaries of the FT-CUP problem.
Definition 3.1 (Connectivity CO). A participant detector satisfies the connectivity property iff the (undirected) graph $G$ is connected.

Proposition 3.1. The Connectivity participant detector is necessary to solve FT-CUP.

Proof. Assume that the resulting graph $G$ returned by the participant detectors is disconnected, i.e. there exist 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.

Definition 3.2 (Full Connectivity FCO). A participant detector satisfies the full connectivity property iff the directed graph $G_{di} = (\Pi, E)$ is such that for all $p, q \in \Pi$, we have $(p, q) \in E$.

Proposition 3.2. The Full Connectivity participant detector is sufficient to solve FT-CUP.

Proof. The processes execute the $\diamond S$ consensus algorithm given in [CT96].

The Information Accuracy and Inclusion properties we have mentioned previously allow processes to query their participant detectors at different times. It is easy to see that if some participant detector $PD$ satisfies the property of $CO$ or $FCO$ if queried at some time $t$, it also satisfies the property when queried at some time $t' > t$. Indeed, adding links to a connected graphs maintains a connected graph and similarly for a fully connected graph. The Information Accuracy property guarantees that the participant detectors truly comply with their specification.

4 Solving CUP (Crash-Free Model)

This section briefly recalls the main results established in [CSS04] for solving CUP in the crash-free model. The authors construct an algorithm that solves CUP based on the One Sink Reducibility participant detector (OSR), which offers both the necessary and sufficient conditions for solving CUP. We first define OSR:

Definition 4.1 (One Sink Reducibility OSR). A participant detector satisfies the one sink reducibility property iff the graph $G$ is connected and the directed acyclic graph obtained by reducing $G_{di}$ to its strongly connected components has one and only one sink.

Figure 1 illustrates the OSR participant detector, by means of two example $G_{di}$ graphs. Recall that the graphs are obtained from the query to the participant detector of each process. The figures also depict the reduction of the $G_{di}$ graphs to their strongly connected components (in dashed lines). Figure 1(a) presents an OSR participant detector: by reducing the graph to its strongly components $A$ through $E$, we obtain a unique sink component, $D$. The participant detector of Figure 1(b) violates the OSR property, since $G_{di}$ reduced to its strongly connected components has two sinks, $A$ and $D$.

We now recall two propositions from [CSS04] that serve as a basis for solving CUP in the crash-free setting.

Proposition 4.1. The One Sink Reducibility participant detector is necessary to solve CUP.

Proof. Suppose that $PD \not\in OSR$, i.e. the directed acyclic graph obtained by reduction of $G_{di}$ 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.

1The same reasoning applies to the OSR participant detector in Section 4. Assume that $PD$ satisfies OSR at time $t$. If the graph $G_{di}$ reduced to its strongly components contains at most one sink, adding edges to $G_{di}$ at time $t'$ cannot increase the number of sinks.

2A sink in a directed graph is a vertex with out-degree 0, i.e. there are no edges leaving the vertex.
Assume there exists an algorithm $A$ 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 $A$ solves CUP.

**Proposition 4.2.** The One Sink Reducibility participant detector is sufficient to solve CUP.

**Proof.** We refer to [CSS04] for an algorithm that solves CUP with $PD \in OSR$.

Clearly, Proposition 4.1 remains true if we allow processes to crash. The $OSR$ participant detector is therefore a natural starting point for solving FT-CUP. However, with process crashes, we can no longer assume that Proposition 4.2 holds. We devote the next section to identifying the necessary and sufficient conditions for which FT-CUP admits a solution.

## 5 Solving FT-CUP (Processes can Crash)

### 5.1 Preliminary Result about Uniform consensus

In traditional networks where the exact set of participating processes $\Pi$ is known to all, [CT96] provide a solution for uniform consensus. Whereas plain consensus requires that two correct processes cannot decide on different values, the uniform variant of consensus specifies that two processes, whether correct or not, cannot decide differently. Hence, if a process decides on a value and crashes, all correct processes must decide on this value. We first show that uniform consensus cannot be solved in our model.

**Proposition 5.1.** Uniform consensus cannot be solved with a participant detector $PD \in OSR$.

**Proof.** By Proposition 4.1 the $OSR$ property is necessary to solve FT-CUP. Consider the finite set $\Pi$ that contains in particular processes $p_1$ and $p_2$. Assume process $p_2$ knows that both itself and process $p_1$ participate in solving the consensus problem. Process $p_1$ on the other hand knows only itself. The resulting $G_{di}$ graph satisfies the $OSR$ property. The local consensus algorithm executed by process $p_1$ must let it decide immediately on its proposed value $v_1$. Indeed, $p_1$ cannot wait for a message from another process; in the case where $\Pi = \{p_1\}$, $p_1$ will wait indefinitely, leading to the violation of the termination property. After deciding, let $p_1$ crash before sending its decision to $p_2$. Process $p_2$ is unable to obtain $p_1$’s decision value and must decide on its own value $v_2$, possibly different than $v_1$, thus violating the uniform agreement property.

Figure 1: Illustration of the $OSR$ property.
In other words, uniform consensus can only be solved in the traditional model where the full set Π is known to all processes. In the remainder of the paper, we consider non-uniform consensus, specified by the validity, agreement and termination properties presented in Section 2.

5.2 Failure Detectors

A synchronous network allows for accurate process failure detection by means of the bounds on the transmission delay of messages and on the process relative speeds. In asynchronous networks there are no such bounds. A direct consequence is the famous FLP impossibility result [FLP85], which states that consensus is not solvable in asynchronous networks, even if only one process can crash. The intuition behind this result is that it is impossible to safely distinguish between a slow process or communication link and a process crash. To circumvent this result, several alternative models have been introduced, such as the partially synchronous network model introduced in [DLS88]. The approach we adopt in this paper is that of [CT96].

In their seminal paper, [CT96] introduce the notion of failure detectors for asynchronous networks. Failure detectors are distributed oracles associated with each process, which, when queried, provide crash suspicion information about the processes. Due to the FLP impossibility result, failure detectors are unreliable — they may provide erroneous information such as mistakenly suspecting a correct process or not suspecting a crashed process. Nevertheless, by satisfying or eventually satisfying well defined properties, the notion of time can be accurately abstracted and consensus can be solved in asynchronous networks. Failure detectors are characterized by variants of the completeness and accuracy properties:

**Strong Completeness** Every crashed process is eventually suspected forever by every correct process.

**Weak Completeness** Every crashed process is eventually suspected forever by (at least) one correct process.

**Strong Accuracy** No process is suspected before it crashes.

**Weak Accuracy** Some correct process is never suspected.

**Eventual Strong Accuracy** There is a time after which correct processes are not suspected by any correct process.

**Eventual Weak Accuracy** There is a time after which some correct process is never suspected by any correct process.

Completeness sets requirements with respect to crashed processes, while accuracy sets requirements with respect to correct processes. In traditional networks, i.e. when Π is known, [CT96] show that the weakest possible properties for solving consensus are Strong Completeness and Eventual Weak Accuracy.

Our context of FT-CUP presents two difficulties, not only can the processes participating in consensus crash, but they are a priori unknown. Therefore, we associate two distributed oracles with each process: a) a participant detector for obtaining information about participating processes and b) a failure detector for obtaining information about crashed processes. Before proceeding, we extend the notion of failure detectors to our model. In [CT96] the accuracy and completeness properties naturally apply to the full set Π, which is known to all processes. When Π is partially known and approximated by participant detectors responsible for accumulating knowledge about participating processes, the accuracy and completeness properties apply on the set of processes returned by the participant detectors. This set is a subset of Π. Naturally, a process can reliably communicate only with the non-suspected processes returned by its participant detector.

To the contrary of traditional networks, we prove in the following proposition that the failure detectors must necessarily satisfy the strong completeness property for solving FT-CUP. Without the full knowledge of Π, some processes in the network may have already decided without notifying Π. To avoid violating the agreement property, a process cannot decide without considering this possibility and must therefore

---

3In terms of not violating the specification of consensus.
communicate with every process returned by its participant detector to retrieve an eventual decision value. To avoid waiting indefinitely for a non suspected process and violating the termination property, a failure detector satisfying strong completeness is necessary.

**Proposition 5.2.** The failure detectors must necessarily satisfy the strong completeness property for solving FT-CUP.

**Proof.** Let $FD$ be a failure detector that does not satisfy strong completeness, i.e. there is (at least) one crashed process that is never suspected by some correct process. Assume the finite set $\Pi$ of processes that contains in particular $p_1$ and $p_2$ such that $PD_{p_1} = \{p_1\}$, $PD_{p_2} = \{p_1, p_2\}$. Process $p_1$, knowing only itself, is unable to exchange messages with other processes in $\Pi$ and must decide on its value $v_1$ in order not to violate the termination property. Process $p_2$, knowing $p_1 \in \Pi$, must either communicate with $p_1$ or suspect it has crashed:

- If process $p_1$ is correct, it may have already decided on its value $v_1$. Process $p_2$ cannot decide differently and must obtain $v_1$ from $p_1$.
- If process $p_1$ has crashed and $p_2$ does not suspect so, it will wait indefinitely for $p_1$’s response, violating the termination property.

We furthermore prove that strong accuracy is also required. The intuition behind the proof is that permitting false suspicions may result in creating two different perceptions of the OSR graph that contain non-intersecting sinks, thus creating a situation where the agreement property can be violated.

**Proposition 5.3.** The failure detectors must necessarily satisfy the strong accuracy property for solving FT-CUP.

**Proof.** Let the processes query only once their participant detector. The $G_{di}$ graph obtained necessarily satisfies the OSR property by Proposition 4.1. By querying their failure detector module, links can be removed or temporarily removed from the $G_{di}$ graph, provided OSR remains satisfied. Assume that the strong accuracy property is not satisfied and correct processes can be falsely suspected. Let $FD_{p_1}(t_1)$ and $FD_{p_2}(t_2)$ be the invocations of a process $p_i$ to its failure detector at times $t_1$ and $t_2$. Without violating the OSR property, the $G_{ds}$ graph from which we remove the set of nodes $FD_{p_1}(t_1)$ and the graph $G_{di}$ from which we remove the set of nodes $FD_{p_2}(t_2)$ contain non-intersecting sinks composed of correct but possibly falsely suspected processes that can decide on different values. The rest of the proof is similar to that of Proposition 4.1 and the agreement property can be violated.

Figure 2: Lack of strong agreement.
Proof. We proceed by contradiction. Assume Proposition 5.4.
Processes must query their participant detector exactly once for FT-CUP to be solvable.

This situation is similar to that of Proposition 5.3.

A safe crash pattern is necessary and sufficient for solving FT-CUP. Propositions 5.2 and 5.3 imply that a perfect failure detector is necessary to solve FT-CUP.

5.3 The Safe Crash Pattern

We define in this section the safe crash pattern, which sets the necessary and sufficient conditions for solving FT-CUP by imposing a restriction on the combination of the participant and failure detectors. The safe crash pattern is the basis upon which we construct the FT-CUP algorithm in Section 5.5.

We now define the safe crash pattern, which regroups the key properties for solving FT-CUP. The pattern’s definition spreads across both the participant detector for discovering processes in II and the failure detector for suspecting crashed processes in II. When there are no process crashes, the $PD \in OSR$ participant detector, defined in Definition 4.1, is necessary to solve CUP. When processes can crash, it is the set of processes discovered by $PD$, from which we remove the set of suspected processes provided by $FD$, that must satisfy the $OSR$ property. This condition constitutes the safe crash pattern and its definition incorporates the results of Propositions 5.2 to 5.4 (Proposition 5.4 is given below).

Definition 5.1 (Safe Crash Pattern). A safe crash pattern is a set of processes such that $PD_{init} - FD_P \in OSR$, where $PD_{init}$ is the single invocation to $PD$ and $FD_P$ is a perfect failure detector.

The safe crash pattern requires that a single invocation be made to the participant detector. This requirement is necessary as we prove in the following proposition. Although the approximation of processes in II can evolve with time, FT-CUP must be solved on a single snapshot of $PD$. If multiple invocations were permitted, the set $PD - FD$ could evolve arbitrarily, creating unrelated $OSR$ snapshots containing different sinks. This situation is similar to that of Proposition 5.3.

Proposition 5.4. Processes must query their participant detector exactly once for FT-CUP to be solvable.

Proof. We proceed by contradiction. Assume $PD$ is queried twice, at times $t_1$ and $t_2$, with $PD(t_1) - FD_P(t_1) \in OSR$ and $PD(t_2) - FD_P(t_2) \in OSR$. These two queries produce two $G_{di}$ graphs, which reduced to their strongly connected components contain two distinct 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. The rest of the proof is similar to that of Proposition 4.1, leading to a contradiction. □

Figure 3 provides an illustration of the result with three processes $p_1$, $p_2$ and $p_3$. Figure 3(a) represents the graph obtained by the first invocation of the processes to their participant detector. Process $p_2$ knows only itself and decides on its value $v_2$. Process $p_3$ communicates with $p_2$, obtains its decision and decides $v_2$. Process $p_3$ now queries its participant detector a second time, discovering process $p_1$ (cf. Figure 3(b)). Process $p_2$ now crashes (cf. Figure 3(c)). Process $p_1$, never having received $p_2$’s decision, now assumes that it is alone in the network and decides on its own value $v_1 \neq v_2$. Agreement is violated. Note that all three graphs satisfy $PD - FD_P \in OSR$.

Proposition 5.5. The safe crash pattern is necessary and sufficient for solving FT-CUP.

Proof.
i) Necessary:

- We know from Proposition 4.1 that $PD \in OSR$ is necessary to solve CUP when there are no crashes and therefore necessary in the more general case when processes can crash.

8
• We know by Propositions 5.2 and 5.3 that a perfect failure detector is necessary.

• We know by Proposition 5.4 that there can only be a single invocation to $PD$ or else the agreement property can be violated.

Hence from the above points, the safe crash pattern as defined in Definition 5.1 is necessary for solving FT-CUP.

ii) Sufficient: The proof is given by providing Algorithm 2 (see page 13) that solves FT-CUP assuming the safe crash pattern.

Whereas failure detectors are unreliable and make mistakes, a participant detector cannot and therefore the safe crash pattern must be verified at all times. Indeed, the one sink reducibility property must be true at all times: if, for a given time $t$, the OSR property is violated, then $G_{di}$ graph reduced to its strongly connected components has two sinks $S_1$ and $S_2$. Processes in $S_1$ and $S_2$ can independently decide on different values, as proved in Proposition 4.1.

The safe crash pattern ensures that any new graph resulting by an additional invocation to the failure detectors (the participant detectors are queried only once) will, when reduced to its strongly connected components, contain a single sink that includes every correct process of every previous sink obtained by previous invocations. This property is not only necessary as we have just shown, but also sufficient to solve FT-CUP as we now prove.

5.4 The Participant Discovery Algorithm

We know from Proposition 5.5 that we must have $(PD_{init} - FD_P) \in OSR$ at any time. Besides this property, we can show that processes in $\Pi$ must also extend the initial knowledge provided by their local participant detector. Indeed, consider $\Pi = \{p_1, p_2\}$ such that $PD_{p_1} = \{p_1, p_2\}$, $PD_{p_2} = p_2$ and $FD_{p_1} = FD_{p_2} = \emptyset$. Although $(PD_{init} - FD_P) \in OSR$, communication channels can be arbitrarily slow and $p_2$ has no way of predicting if it is alone or not in the network. Process $p_2$ must therefore decide on its own in order to not violate the termination property. Now that $p_2$ has decided, the only way for it to propagate the decision to $p_1$’s existence, e.g. by receiving a message from it.

To augment the initial knowledge about other participants such that FT-CUP is solvable, processes execute Algorithm 1, a fault-tolerant token-based discovery algorithm. Every process $p_i \in \Pi$ queries its participant detector and executes $\text{discover\_participants()}$. Notice that processes query their participant detector only once. Every token generated carries the identifier of its creator ($\text{token}_i.\text{issuer}$), the set of processes already visited by the token ($\text{token}_i.\text{visited}$) and the set of participants discovered but not yet visited by the token ($\text{token}_i.\text{tovisit}$).

Prior to forwarding the token, $p_i$ adds to $\text{token}_i.\text{tovisit}$ the processes it can communicate with, i.e. processes returned by its local participant detector (line 4). The token is then forwarded to any process present in $\text{token}_i.\text{tovisit}$ (line 9). Upon reception of $\text{token}_i$ by a process $p_j$, $p_j$ checks whether it is the issuer of $\text{token}_i$. If yes, the algorithm terminates and returns the set of visited processes (line 13). If not, it updates the data structures stored in $\text{token}_i$ as follows. First, it adds to $\text{token}_i.\text{tovisit}$ all processes $p_j$
knows that have not yet been visited and that are not suspected by \( p_j \). Then \( p_j \) removes its own ID from \( \text{token}_i\_tovisit \) and adds it in \( \text{token}_i\_visited \). If there are no more processes to visit, \( p_j \) sends the token back to \( \text{token}_i\_issuer \). Otherwise, it simply forwards \( \text{token}_i \) to any one of them. Considering that any process can crash as long as \((PD_{init} - FD_P) \in OSR\) holds and that by strong completeness \( FD_P \) only eventually suspects crashed processes, a token can be forwarded to a crashed process resulting in its loss. To circumvent this, the algorithm continuously generates new tokens. By the strong completeness property of the failure detector, all crashed participants will eventually be suspected (line 15) and at least one token is guaranteed to return.

We denote by \( \text{participants}_i \) the knowledge a process \( p_i \in \Pi \) has about other participating processes after the execution of Algorithm 1. We have directly:

\[
PD_{init} \subseteq \text{participants}_i.
\]

Additionally, provided the safe crash pattern \((PD_{init} - FD_P) \in OSR\), the set \( \text{participants}_i \) satisfies the following properties :

\begin{enumerate}
  \item Property 5.1. The \( \text{participants}_i \) sets have a non-empty intersection.
  \[
  \bigcap_{p_i \in \Pi} \text{participants}_i \neq \emptyset
  \]
  \item Property 5.2. Every process in the intersection knows exactly the set processes in the intersection.
  \[
  \text{participant}_i = \bigcap_{p_j \in \Pi} \text{participants}_i, \forall p_i \in \bigcap_{p_j \in \Pi} \text{participants}_i
  \]
\end{enumerate}

\begin{algorithm}
1 \( \text{token}_i\_issuer \leftarrow p_i \);
2 \( \text{token}_i\_visited \leftarrow \emptyset \);
3 \( \text{token}_i\_tovisit \leftarrow \emptyset \);

  //Single invocation to \( PD_{p_i} \)
4 \( \text{neighbors}_i \leftarrow PD_{p_i} \);
5 \text{discover\_participants}() : 
6 \( \text{token}_i\_visited \leftarrow \{ p_i \} \);
7 \( \text{token}_i\_tovisit \leftarrow \text{neighbors}_i \setminus \{ p_i \} \);
8 \text{while} true \text{ do}
9 \quad \text{send} \text{token}_i \text{ to any} \ p_j \in \text{neighbors}_i \setminus \{ p_i \} ;
10 \quad \text{wait}(\text{timeout}) ;
11 \text{Upon receive}(\text{token}_j) \text{ from} \ p_k : 
12 \quad \text{if} \ \text{token}_j\_issuer = p_k \text{ then}
13 \quad \quad \text{return} \text{token}_j\_visited ;
14 \quad \quad \{ \text{Algorithm terminates} \}
15 \quad \text{token}_j\_tovisit \leftarrow (\text{token}_j\_tovisit \cup (\text{neighbors}_i \setminus \text{token}_j\_visited)) \setminus \text{FD}_{p_i} ;
16 \quad \text{token}_j\_tovisit \leftarrow \text{token}_j\_tovisit \setminus \{ p_i \} ;
17 \quad \text{token}_j\_visited \leftarrow \text{token}_j\_visited \cup \{ p_i \} ;
18 \quad \text{if} \ \text{token}_j\_tovisit = \emptyset \text{ then}
19 \quad \quad \text{send} \text{token}_j \text{ to} \text{token}_j\_issuer ;
20 \quad \text{else}
21 \quad \quad \text{send} \text{token}_j \text{ to any} \ p_l \in \text{token}_j\_tovisit ;
\end{algorithm}

5.4.1 Correctness of Algorithm 1

To prove the algorithm’s correctness, we must show that it terminates and satisfies Properties 5.1 and 5.2 stated above. The algorithm terminates when \( \text{token}_i \) (the token issued by \( p_i \)) has returned to \( p_i \). Let us first
consider the case without crashes and 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 reachable from \( p_i \) are visited, in which case line 19 is executed and eventually \( p_i \) executes line 13. If processes can crash, a token might be forwarded to a failed process resulting in the loss of the token. By the strong completeness property of \( FD \in P \), every crashed process will eventually be suspected and removed from \( token_j.tovisit \) (line 15). Since new tokens are continuously generated (lines 8-10) by \( p_i \) and the number of failures is bounded, a token will eventually return to \( p_i \).

To prove that Algorithm 1 satisfies Property 5.1, we show that the intersection of the \( participants \) sets is not empty. We assume that \( (PD_{init} - FD_P) \in OSR \) holds at any time and denote by \( S_0 \) the single sink of that OSR at time \( t_0 \). By the definition of the OSR property, we know that a path exists from every process in \( \Pi \) to every process in \( S_0 \). If no process in \( S_0 \) crashes, then \( S_0 \) is the non-empty intersection. If some process in \( S_k \) crashes at time \( t_k \), then the sink changes to \( S_{k+1} \). Either \( S_{k+1} \) is a subset of \( S_k \) or all processes in \( S_{k+1} \) have crashed. In that latter case, \( S_{k+1} \) is a subset of \( participants \) for all \( p_i \in \Pi \) because \( (PD_{init} - FD_P) \) remains OSR and no new links have been added to the graph since the unique initial invocation to \( PD \).

By definition of a sink in an OSR graph, processes belonging to the sink \( S \) form a strongly connected component without any outgoing link. Since a path connects any pair of processes in \( S \), the participants in the sink discover exactly all correct processes in \( S \) and Property 5.2 is satisfied.

5.5 The FT-CUP Algorithm

This section introduces informally FT-CUP (cf. Algorithm 2) and provides a general intuition on how consensus is reached among all participants in spite of process crashes. Every process \( p_i \in \Pi \) has two oracles: a participant detector \( PD_{p_i} \in OSR \) and a failure detector \( FD_{p_i} \in P \). The algorithm assumes that \( (PD_{init} - FD_P) \) is OSR at any time. Firstly, processes execute the fault-tolerant discovery algorithm pre-
sented in Algorithm 1 to augment their initial knowledge about processes in II (line 7). The single invocation to \(PD\) is made at the beginning of Algorithm 1. To handle process crashes, FT-CUP implements a background task that is executed each time a process is suspected to have failed (lines 15-20). It is responsible for removing crashed processes from the \(participants\) set and backtrack in the algorithm when necessary. For clarity reasons, we first consider that processes do not fail. Situations where processes crash are presented later in this section.

The key strategy is to ensure that correct processes belonging to the sink impose their decision to the other processes. After the execution of Algorithm 1, every process \(p_i\) elects as a leader the process in \(participants_i\) with the lowest identifier (line 9). Figure 4(a) depicts the situation after the execution of the token discovery algorithm. Processes are represented as circles. The boxes, attached to every process, show the \(participants\) sets discovered. Processes in strongly connected component A will elect \(p_1\) as their leader, processes in B will elect \(p_2, p_3\) will be its own leader and \(p_7\) the leader for processes in components D and E. In Figure 4, leaders are represented with darker circles. Non leader processes then register themselves by sending a \(decisionRequest\) message to their respective leader to get the \(estimate\) (line 14). In the next phase, the leaders must identify among themselves the leader of the sink component. In order to do so, each leader sends the \(am I the sink leader?\) message to all the processes in its \(participants\) set (line 12), and waits for acknowledgments. Upon receiving \(am I the sink leader?\) from a leader \(p_1\), a process \(p_i\) either responds with \(lack\) if \(p_i\)'s leader is \(p_1\) (line 26), or otherwise with \((lnack, leader_i)\), where \(leader_i\) is \(p_i\)'s leader (line 23). Since all participants will have discovered the sink component D (by Property 5.1), \(p_7\) will be the only leader to receive only \(lacks\) (line 28). Other leaders will receive \((lnack, leader_j)\) messages from processes in D and will send a \(estimateRequest\) message to \(leader_j\) (line 35). Upon reception of this message the leader either sends its \(estimate\) (line 38) if it has already decided or registers the request using the set \(estimateRequestors_i\) (line 40). Finally, the sink leader decides on its own \(estimate\) value (line 30) and sends it to all the processes in the \(decisionRequestors\) set, reaching all non sink leaders. The non sink leaders propagate the \(estimate\) to the processes registered in their \(decisionRequestors\) set. The \(estimate\) propagates across processes via the \(decisionRequestors\) sets, eventually to all processes. Upon reception of a recent \(estimate\) and prior to deciding, a process \(p_i\) send a \(doc(estimate_i)\) (decide on \(estimate_i\)) message (line 44) to check whether \(estimate_i\) has been adopted by all processes in \(participants_i\). The exact meaning of recent \(estimate\) is given in the next paragraph. If a process \(p_i\) receives a \(doc\) and has already decided or knows a more recent \(estimate\) it replies with a \((enack, estimate_i)\) (lines 47 and 50) or otherwise with \(eack\) (line 52). Since the \(participants\) sets have a non-empty intersection (by Property 5.1), two processes cannot receive only \(eack\) messages for two different \(estimates\). Finally, upon reception of \(eack\) messages from all processes in \(participants_i\), \(p_i\) decides.

We now investigate two typical scenarios where processes failure have an impact on the algorithm. In Figure 4(b), the sink leader \(p_7\) crashes. The strong completeness property of \(FD \in \mathcal{P}\) ensures that all processes waiting for an estimate from \(p_7\) will eventually suspect it (line 17) and execute again \(propose\) to elect a new leader (line 18). In our example, processes in component D and E will choose \(p_8\) as their new leader. But since the failure detection mechanism is eventual, participants do not suspect crashed process at the same time. This situation may lead to the coexistence of several \(estimate\) values in the network out of which only one is valid. Hence, processes need a means to decide if an \(estimate\) is more recent than an other. To achieve this, an \(estimate\) is defined as a couple \((estimate_i, p_i)\), where \(p_i\) is the process which proposed \(estimate_i\). It follows that \((estimate_i, p_i)\) is more recent than \((estimate_j, p_j)\) if \(p_i > p_j\). is. If further processes crashes, it may happen that the sink disappears completely, like in Figure 4(c). In such a case, by the safe crash pattern there must exist a new single sink known by all participants. In our example, component C is the new sink.

5.5.1 Correctness of Algorithm 2

To prove that Algorithm 2 is correct, we show that it satisfies the three properties of Termination, Agreement and Validity presented in Section 2.
Algorithm 2: Solving consensus with $PD \in OSR$, $FD \in P$ and the safe crash pattern for a process $p_i \in \Pi$

1. \text{/*leader estimate*/}
2. \text{leader}, ← 1;
3. \text{/*set of leaders*/}
4. \text{leaders}, ← \emptyset;
5. \text{/*set of leaders disagreed upon*/}
6. \text{lnackedLeaders}, ← \emptyset;
7. \text{/*set of processes requiring a notification of the decision*/}
8. \text{estimateRequestors}, ← \emptyset;
9. \text{/*initial value*/}
10. \text{estimate}, ← 1;
11. \text{/*decision value*/}
12. \text{decision}, ← 1;
13. \text{/*execute and store result from Algorithm 1, using $PD \in OSR$*/}
14. \text{participants}, ← discover_participants();
15. \text{propose($v_i$);}
16. \text{leader}, ← \min(\text{participants});
17. \text{if } p_i = \text{leader}, then
18. \text{\hspace{1em} /*the process may be the sink leader, send a message*/}
19. \text{\hspace{1em} estimate}, ← ($v_i, p_i$);
20. \text{\hspace{1em} send am I the sink leader? to all $p \in \text{participants}$;}
21. \text{else}
22. \text{\hspace{1em} send estimateRequest to leader;}
23. \text{\hspace{1em} /*the procedure exits and the process now handles the following events*/}
24. \text{Upon $p_i \in FD$;}
25. \text{\hspace{1em} /*process $p_i$ has crashed, update required data structures*/}
26. \text{participants}, ← \text{participants}, \setminus \{p_i\};
27. \text{if } p_i = \text{leader, then}
28. \text{\hspace{1em} /*$p_i$ is not just any process, but the leader estimate*/}
29. \text{\hspace{1em} propose($v_i$);}
30. \text{\hspace{1em} if leader, \in lnackedLeaders, then}
31. \text{\hspace{1em} /*Previously nacked local leader is now believed to be sink leader*/}
32. \text{\hspace{1em} send lack to leader;}
33. \text{else}
34. \text{\hspace{1em} /*$p_i$ sends a leader ACK*/}
35. \text{\hspace{1em} send lack to $p_i$;}
36. \text{Upon reception of am I the sink leader? from process $p_j$;}
37. \text{if } p_j \neq \text{leader}, then
38. \text{\hspace{1em} /*disagreement on leader identity; $p_i$ sends a leader NACK*/}
39. \text{\hspace{1em} send (lack,leader,) to $p_j$;}
40. \text{\hspace{1em} lnackedLeaders, ← lnackedLeaders, \cup \{p_j\};}
41. \text{else}
42. \text{\hspace{1em} /*$p_i$ sends a leader ACK*/}
43. \text{\hspace{1em} send lack to $p_j$;}
44. \text{Upon reception of lack from process $p_j$;}
45. \text{if lack received from $\forall p \in \text{participants}$, then}
46. \text{\hspace{1em} /*the process is indeed the sink leader; propagate estimate, as the decision*/}
47. \text{\hspace{1em} decision, ← estimate,.value;}
48. \text{\hspace{1em} decide(decision,);}
49. \text{\hspace{1em} send estimate, to all $p \in \text{estimateRequestors};$}
50. \text{Upon reception of (lnack,leader,) from process $p_j$;}
51. \text{if leader, \notin \text{leaders, then}
52. \text{\hspace{1em} /*request an estimate from leader*/}
53. \text{\hspace{1em} leaders, ← leaders, \cup \{leader,\};}
54. \text{\hspace{1em} send estimateRequest to leader;}
55. \text{Upon reception of estimateRequest from process $p_i$;}
56. \text{if decision, \neq 1, then}
57. \text{\hspace{1em} send estimate, to $p_j$;}
58. \text{else}
59. \text{\hspace{1em} send estimateRequestors, ← estimateRequestors, \cup \{p_j\};
60. \text{Upon reception of estimateRequest from process $p_i$;}
61. \text{if estimate, process \neq p_i, or}
62. \text{\hspace{1em} estimate,.process > estimate,.process then}
63. \text{\hspace{1em} estimate, ← estimate,;
64. \text{\hspace{1em} send doc(estimate,) to all $p_j \in \text{participants};$}
65. \text{Upon reception of doc(estimate,) from process $p_i$;}
66. \text{if decision, = estimate,.value then}
67. \text{\hspace{1em} send each to $p_i$;}
68. \text{\hspace{1em} else if estimate,.process = p_i, or}
69. \text{\hspace{1em} estimate,.process > estimate,.process then}
70. \text{\hspace{1em} estimate, ← estimate,;
71. \text{\hspace{1em} send each to $p_i$;}
72. \text{\hspace{1em} else}
73. \text{\hspace{1em} send (lnack,estimate,) to $p_j$;}
74. \text{Upon reception of each from process $p_j$;}
75. \text{if each received from all $p_k \in \text{participants}$, then}
76. \text{\hspace{1em} decision, ← estimate,.value;}
77. \text{\hspace{1em} decide(decision,);}
78. \text{\hspace{1em} send estimate, to all $p \in \text{estimateRequestors};$}
79. \text{Upon reception of (lnack,estimate,) from process $p_i$;}
80. \text{\hspace{1em} estimate, ← estimate,;
81. \text{\hspace{1em} send doc(estimate,) to all $p \in \text{participants};$}
Termination  FT-CUP terminates, for a participant, upon execution of decide at line 30 or line 56. By Property 5.2 of Algorithm 1 we know that processes in the sink have discovered exactly all processes belonging to the sink (line 7). Hence, they all elect the same leader \( p_l \), i.e. the process with the lowest identifier (line 9). If \( p_l \) is not correct and crashes, then it will be eventually suspected by all processes in \( \text{participants}_l \) (line 17) and processes will re-execute propose to elect a new leader.

Consider that \( p_l \) is correct and never crashes. It sends a message \( \text{am I the sink leader?} \) to all processes in \( \text{participants}_l \) (line 12). Since all processes in \( \text{participants}_l \) agree on \( p_l \) leadership (line 26), \( p_l \) receives a \( \text{ack} \) message from all processes in \( \text{participants}_l \) non-suspected by \( p_l \) (line 28). Then \( p_l \) decides on its estimate (line 30) and sends the pair \((\text{estimate}_l, p_l)\) to all processes in \( \text{estimateRequestors}_l \). Notice, that no other leader \( p_i \neq p_l \) can decide before \( p_l \) since any \( p_i \) will receive at least one \((\text{lnack}, p_l)\) message from all processes located in the sink. Upon reception of a \((\text{lnack}, p_l)\), local leaders send a \( \text{estimateRequest} \) to \( p_l \) to get the \( \text{estimate} \) from \( p_l \).

Due to the crash of previous leaders, several estimates may coexist at the same time. If a process \( p_j \) has already received an \( \text{estimate}_j \) from \( p_j \) earlier, \( p_j \) replaces \( \text{estimate}_j \) with \( \text{estimate}_l \) only if it has been issued by a more recent leader, i.e. \( p_l > p_j \). Upon reception of \( \text{estimate}_j \), \( p_l \) tries to impose it to all processes it knows by sending a \( \text{do}c(\text{estimate}_j) \) (decide on \( \text{estimate}_j \)) message to all \( p_j \in \text{participants}_l \) (line 44). Then a process \( p_l \) either reply \((\text{enack}, \text{estimate}_i)\) to \( p_i \) if it \( \text{estimate}_i \) is newer than \( \text{estimate}_j \) or \( \text{enack} \) otherwise. If \( p_l \) receives only \( \text{ack} \) messages, then it can safely decide (line 56) or otherwise \( p_l \) adopts \( \text{estimate}_j \) and sends again a \( \text{do}c(\text{estimate}_j) \).

Agreement  We prove that the agreement property is satisfied by contradiction. Assume that a first sink leader \( p_k \) has decided on \( \text{estimate}_k \) and then crashed. Assume also that the new leader \( p_l \) has decided on \( \text{estimate}_l \neq \text{estimate}_k \). Both \text{estimates} can be different since FT-CUP solves the non-uniform variant of consensus. We suppose that there exist two correct processes \( p_i \) and \( p_j \) that have decided on \( \text{estimate}_k \) and \( \text{estimate}_j \) respectively. This implies that both \( p_i \) and \( p_j \) have received only \( \text{ack} \) messages from every process in \( \text{participants}_i \) and \( \text{participants}_j \) to acknowledge their respective and different estimate. Since any sink leader must belong to the \text{participant} set of every process (by Properties 5.1 and 5.2), \( p_i \) has received an \( \text{ack} \) from \( p_l \) for \( \text{estimate}_k \) and therefore \( \text{estimate}_l = \text{estimate}_k \).

Validity  Validity is trivially satisfied since the decision is the \text{estimate} of some process.

6  Summary and Future Work

We have considered the problem of reaching agreement in mobile self-organized networks such as MANETs. To this means we have specified FT-CUP, a derived consensus problem in asynchronous networks where the participating process are unknown and can crash. This specification captures the characteristics of highly dynamic self-organized networks. We have adapted the failure detector definition of [CT96] to the case where \( \Pi \) is unknown: the accuracy property does not apply to the entire set \( \Pi \) but to the subset of \( \Pi \) that each participating process are unknown and can crash. This specification captures the characteristics of highly mobile ad hoc networks, for smaller and less dynamic MANETs it can be interesting to explore...
the partially synchronous network model. Partial synchrony [DLS88] assumes that upper bounds to message delivery delay or relative process speeds exist, but are either unknown or hold eventually. Similarly to classic consensus in traditional networks, partial synchrony may lead to more flexible conditions for designing an FT-CUP algorithm tailored for particular MANET scenarios.

References


