On the Minimal Knowledge Required for Solving Stellar Consensus

Robin Vassantlal, Hasan Heydari, and Alysson Bessani
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{rvassantlal, hheydari, anbessani}@ciencias.ulisboa.pt

***Abstract*—Byzantine Consensus is fundamental for building consistent and fault-tolerant distributed systems. In traditional quorum-based consensus protocols, quorums are defined using globally known assumptions shared among all participants. Motivated by decentralized applications on open networks, the Stellar blockchain relaxes these global assumptions by allowing each participant to define its quorums using local information. A similar model called Consensus with Unknown Participants (CUP) studies the minimal knowledge required to solve consensus in ad-hoc networks where each participant knows only a subset of other participants of the system. We prove that Stellar cannot solve consensus using the initial knowledge provided to participants in the CUP model, even though CUP can. We propose an oracle called *sink detector* that augments this knowledge, enabling Stellar participants to solve consensus.**

***Index Terms*—Byzantine Consensus, Blockchain, Quorum Systems, Consensus with Unknown Participants, Stellar.**

I. Introduction

Consensus is a fundamental building block for distributed systems that remain available and consistent despite the failure of some participants [1]–[3]. In this problem, participating processes agree on a common value from the initially proposed values. This problem was extensively studied considering different synchrony assumptions (e.g., partially synchronous) and failure models (e.g., Byzantine failures) in the permissioned setting, where the set of participants and the fault threshold is known *a priori* by all participants (e.g., [3]–[6]).

The Nakamoto consensus protocol [7], used in Bitcoin, makes it possible to solve consensus without having a single global view of the system. In Nakamoto consensus, the set of all participants is unknown, and the system's fault tolerance is determined based on the total amount of computing power controlled by the adversary. Even though this protocol opens doors for anyone to participate in consensus and is scalable in the number of participants, its performance is orders of magnitude lower than consensus protocols for the permissioned setting [8].

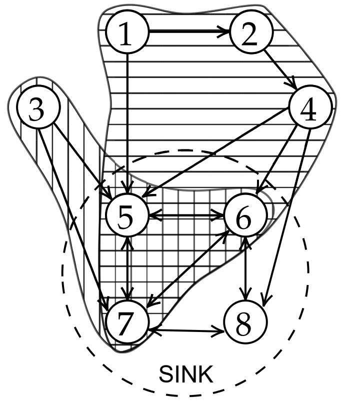
However, with the popularization of blockchains, the demand for scaling consensus to many participants while maintaining high performance led researchers to propose interesting alternatives. Examples include hybrid consensus [9]–[11] and asymmetric trust-based protocols [12]–[14]. In the former, a committee of participants is randomly selected from a network of unknown size in proportion to the resources they control (e.g., computing power or stake) to execute a traditional permissioned Byzantine fault-tolerant consensus (e.g., PBFT [3]). In the latter, the global knowledge about the system membership and fault threshold required in the permissioned consensus is relaxed by enabling each participant to declare a partial view of the participants it can trust. This paper focuses on one of the most well-known asymmetric trust-based protocols called Stellar [13], [15].

The Stellar blockchain enables exchanging digital assets worldwide without relying on centralized authorities such as banks. Stellar comprises two main elements: a Byzantine quorum-based consensus protocol called SCP (Stellar Consensus Protocol) and a network of trust called the Stellar network. SCP maintains a consistent ledger of transactions where participants neither need to know all participants nor the maximum number of participants that can fail in partially synchronous systems. Besides, it allows anyone to join the network without reconfiguring the system.

SCP is executed on the Stellar network built using trust relationships declared by each participant. More specifically, at the beginning of the execution, each participant in Stellar has only access to a set of slices, where each slice is a set of participants. Even though it is unclear how slices are defined in Stellar [15], in practice, these slices are manually defined based on a list of trusted participants. The combination of these slices forms quorums. SCP can solve consensus in the Stellar network if quorums satisfy a property called consensus cluster [16].

There is a similar line of research on a model called Consensus with Unknown Participants (CUP) that studies the knowledge required to solve consensus in settings in which each participant joins the network knowing only a subset of other participants and the fault threshold of the system [17]–[20]. In this model, each participant's knowledge about the existence of other participants is encapsulated in a local oracle called *participant detector*. The union of the information the participant detectors provide forms a knowledge connectivity graph, where a vertex is a participant, and an edge represents the knowledge between two participants. For example, in Fig. 1, participant 1 initially knows participants 2 and 5.

The CUP model allows the establishment of the minimal knowledge necessary and sufficient under specific synchrony and fault assumptions to solve consensus. The knowledge requirement increases as the synchrony assumptions are relaxed and the fault assumptions get stronger. For example, each participant in Byzantine Fault-Tolerant (BFT) CUP [17] requires more knowledge than in the fault-free CUP model [18].



Participants' knowledge:

$PD_1 = \{2, 5\}$
$PD_2 = \{4\}$
$PD_3 = \{5, 7\}$
$PD_4 = \{5, 6, 8\}$
$PD_5 = \{6, 7\}$
$PD_6 = \{5, 7, 8\}$
$PD_7 = \{5, 6, 8\}$
$PD_8 = \{6, 7\}$

Fig. 1. An example of a knowledge connectivity graph with 8 participants. For each participant $i \in \{1, 2, \ldots, 8\}$, PD_i shows the information provided by its participant detector, i.e., the knowledge of i. Participants 5, 6, 7, and 8 form the sink component.

The main question we address in this paper is to determine whether SCP can be executed with the minimal knowledge established in the CUP model, i.e., can each participant define its slices using only a list of participants and a fault threshold? We present two attempts to answer this question. The first attempt locally defines slices for each participant using only a list provided by its participant detector and the fault threshold. We prove that this information is not enough for SCP. The second attempt successfully defines slices using some extra information obtained by increasing the knowledge of each participant. We indeed define and implement an oracle called *sink detector* that provides such required extra information. These results show that, differently from the BFT-CUP protocol [17], SCP is not powerful enough to solve consensus under minimal proven initial knowledge conditions; however, this limitation can be circumvented using a sink detector.

Contributions. This paper makes the following contributions:

- We show that SCP cannot solve consensus when each participant has only the minimum knowledge required to solve consensus.
- We propose an oracle – *sink detector* – by which participants can solve consensus using SCP when each participant has access only to the same knowledge required to solve CUP.

Paper organization. The remainder of the paper is organized as follows. Section II presents the related work. Section III presents our system model and describes the background of this paper. Section IV shows that SCP cannot solve consensus when each participant is given the same knowledge required by CUP. Section V shows how slices can be defined using a participant detector, the fault threshold, and a sink detector. Section VI presents the implementation of the sink detector. Finally, Section VII concludes the paper.

II. Related Work

Stellar. The design of protocols for participants with different trust assumptions (i.e., participants can trust different sets of participants) was first studied in [21]. Ripple [14], [22] attempts to use this approach to solve consensus in the permissionless setting, with the goal of establishing an efficient blockchain infrastructure; however, the goal is not completely attained due to existing safety and liveness violations [23]. Stellar [13], [15], which is based on the Federated Byzantine Quorum System (FBQS) (formally studied afterward in [12]), successfully achieves that goal. In this new attempt, a network of trust emerges from the partial view declared by each participant. Solving consensus in this network is guaranteed if it satisfies the *intact set* property, which states that all correct participants must form a quorum, and any two quorums formed by correct participants must intersect.

A connection between FBQS and the dissemination Byzantine quorum systems [24] was established in [12] and [25], demonstrating how to construct a dissemination Byzantine quorum system that corresponds to an FBQS. Later work by Losa et al. [16] generalized FBQS to Personal Byzantine Quorum System (PBQS). They proved that solving consensus with weaker properties than the intact set is possible using a *consensus cluster*. Specifically, forming a quorum by all correct participants is not required in the consensus cluster. Cachin and Tackmann [26] extended Byzantine Quorum Systems (BQS) [24] from the symmetric trust model to the asymmetric model, which made it easier to compare PBQS with the classical BQS model. Recently, Cachin et al. generalized the asymmetric trust model by allowing each participant to make assumptions about the failures of participants it knows and, through transitivity, about failures of participants indirectly known by it [27].

Consensus with Unknown Participants (CUP). The Consensus with Unknown Participants (CUP) problem has been developed through various steps, adapting to different system models. Initially, the problem was defined by Cavin et al. [18] for failure-free asynchronous systems by introducing the participant detector abstraction to provide participants with initial information about the system membership. The information provided to participants collectively forms a knowledge connectivity graph. This work determines a knowledge connectivity graph's necessary and sufficient properties to solve CUP. Later, the CUP problem was solved in [19] for crash-prone systems enriched with the Perfect ($\mathcal{P}$) failure detector [28]. As $\mathcal{P}$ requires synchrony, Greve and Tixeuil [20] relaxed the assumption to partial synchrony [4] by increasing the minimum required knowledge, i.e., increasing the connections in the knowledge connectivity graph (detailed in the next section), which they show to be minimum to tolerate crash failures without requiring synchrony. The last milestone expanded CUP to tolerate Byzantine failures by introducing the BFT-CUP protocol [17], [29].

More recently, it was shown that synchrony is required to solve consensus without knowing the number of participants and the fault threshold [30].

Sleepy model. Both Stellar and CUP solve consensus in partially synchronous systems, where participants can be correct or faulty. It is also assumed that correct members participate throughout the whole execution of the protocols, which is unrealistic in practice. In the recently proposed sleepy model [31], [32], participants in a synchronous system are assumed to be either awake or asleep, whereas awake participants can be faulty or correct. In this model, the system's fault tolerance dynamically changes as the participants transition between awake and asleep states. Further, consensus can be solved if the majority of awake participants are correct at any time. Differently from Stellar and CUP, in this model, all participants know the set of participants in the system.

Consensus in directed graphs. Somewhat similar to CUP, several works study consensus in directed graphs, e.g., [33]–[36]. However, those works study the requirements of the underlying communication graph to solve consensus under different assumptions. For example, Tseng and Vaidya [36] proved the minimal conditions of the underlying communication graph, where a participant i can send messages to participant j if there is a directed edge from i to j in the graph; otherwise, i cannot send messages to j. Typically, these works assume the set of participants, and the underlying communication graph is known by all participants. In the CUP model considered in this paper, the communication graph is complete, and the goal is to study the required initial knowledge about other participants, which forms a knowledge connectivity graph (see Fig. 1), to solve consensus without knowing the total number of participants in the system.

III. PRELIMINARIES

A. System Model

We consider a partially synchronous distributed system [4] in which the network and processes (also called participants) may behave asynchronously until some *unknown* global stabilization time GST after which the system becomes synchronous, with *unknown time bounds for computation and communication*. This system is composed of a finite set Π of processes drawn from a larger universe $\mathcal{U}$. In a known network, Π is known to every process. In contrast, in an unknown network, process $i \in \Pi$ knows only a subset $\Pi_i \subseteq \Pi$. We assume that Π is static during the execution of an instance of consensus, i.e., no process leaves or joins the system. Our analysis is for a single instance of consensus.

Processes are subject to Byzantine failures [1]. A process that does not follow its algorithm is called *faulty*. A process that is not faulty is said to be *correct*. During the execution of an instance of consensus, we denote W as the set of processes that remain correct and define $F = \Pi \setminus W$ as the set of processes that can fail. We consider a *static Byzantine adversary*, i.e., F is fixed at the beginning of the protocol execution by the adversary. Even though W and F are unknown, $|F| \leq f$, where f is the maximum number of faulty processes. We assume that f is known unless stated otherwise.

We further assume that all processes have a unique id, and it is infeasible for a faulty process to obtain additional ids to launch a *Sybil attack* [37]. Processes communicate by message passing through authenticated and reliable point-to-point channels. 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 can send messages to it.

B. The Consensus Problem

In the consensus problem, each process i *proposes* a value v_i, and all correct processes *decide* the same value v among the proposed values. Formally, any protocol that solves consensus must satisfy the following properties (e.g., [38]):

- *Validity:* 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.
- *Integrity:* every correct process decides at most once.

C. Byzantine Quorum Systems

Byzantine Quorum Systems (BQS) enable solving consensus despite Byzantine failures [39]. A BQS is composed of a set of quorums $\mathcal{Q}$, where each quorum $Q \in \mathcal{Q}$ is a subset of processes that satisfies two properties:

- *Consistency*: every two quorums intersect in at least one correct process, i.e., $\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \cap W \neq \emptyset$.
- *Availability*: there is at least one quorum composed only by correct processes, i.e., $\exists Q \in \mathcal{Q} : Q \subseteq W$.

If one of these two properties is not satisfied, the correctness of a consensus protocol based on BQS cannot be guaranteed. Furthermore, in BQS, Π and f are known for every process. Notice that quorums in a BQS are shared among processes, i.e., if a set of processes $Q \subseteq \Pi$ is a quorum in a BQS, then Q is a quorum for every process in Π.

D. Stellar Model

The Stellar model relaxes the global knowledge assumption about Π and f by employing Federated Byzantine Quorum System (FBQS) [12], [13], [15]. In FBQS, at the beginning of the execution, each process i has only access to its quorum *slices*, which are simply referred to as slices. Each slice of a process i is a set of processes that i trusts. We denote the set of all slices of a process i by $\mathcal{S}_i$. Given a set $A \subseteq \Pi$, we define $\mathcal{S}_A = \{\mathcal{S}_i \mid \forall i \in A\}$, and for each $i \in A$, $\mathcal{S}_A[i]$ equals $\mathcal{S}_i$. We consider that the union of all slices of i is Π_i, i.e., $\bigcup_{S \in \mathcal{S}_i} S = \Pi_i$.

Definition 1 (Quorum)**.** A set of processes Q is a quorum if each process $i \in Q$ has at least a slice contained within Q, i.e., $\forall i \in Q, \exists S \in \mathcal{S}_i : S \subseteq Q$.

Algorithm 1 Determining if a set Q is a quorum.

function $\texttt{is\_quorum}(Q, \mathcal{S}_Q)$
for all $i \in Q$
  if $\nexists S \in \mathcal{S}_Q[i] : S \subseteq Q$
    return *false*
return *true*

We say that quorum Q is a quorum for a process i if i belongs to it and it contains at least a slice of i. We denote the set of all quorums of a process i by $\mathcal{Q}_i$. Each process i attaches $\mathcal{S}_i$ to all of the messages it sends so that any other process knows $\mathcal{S}_i$ by receiving a message from i. We introduce a function $\texttt{is\_quorum}(Q, \mathcal{S}_Q)$ (Algorithm 1) by which a process i can identify whether Q is a quorum using $\mathcal{S}_Q = \{\mathcal{S}_j \mid \forall j \in Q\}$.

In the Stellar model, correct processes can solve consensus under partial synchrony if quorums form a *consensus cluster* (Definition 3). A consensus cluster emerges if quorums are *intertwined* as defined below. The following three definitions are adapted from [16].

Definition 2 (Intertwined)**.** A set I of correct processes is intertwined if, for any two members i and j of I, the intersection of any quorum Q of i and any quorum Q' of j contains at least one correct process, i.e., $\forall i, j \in I, \forall Q \in \mathcal{Q}_i, \forall Q' \in \mathcal{Q}_j : Q \cap Q' \cap W \neq \emptyset$.

Definition 3 (Consensus cluster)**.** A subset $I \subseteq W$ of the correct processes is a consensus cluster when:

- *Quorum Intersection*: I is intertwined, and
- *Quorum Availability*: if $i \in I$ then there is a quorum Q of i such that every member of Q is correct and is inside I, i.e., $Q \subseteq I$.

The quorum intersection property allows to guarantee the agreement and integrity property of consensus, while quorum availability ensures that every correct process makes progress by having at least a quorum composed entirely of correct processes, i.e., it enforces the termination property of consensus.

Definition 4 (Maximal consensus cluster)**.** A maximal consensus cluster is a consensus cluster that is not a strict subset of any other consensus cluster.

All correct processes of the system can solve consensus if there is exactly one maximal consensus cluster C in the system such that $C = W$ [16]. To see an example of slices and quorums, consider the graph depicted in Fig. 1. In this graph, we assume that $W = \{1, 2, \ldots, 7\}$ and $F = \{8\}$. Besides, let slices of each correct process be defined as follows: $\mathcal{S}_1 = \{\{2, 5\}\}$, $\mathcal{S}_2 = \{\{4\}\}$, $\mathcal{S}_3 = \{\{5, 7\}\}$, $\mathcal{S}_4 = \{\{5, 6\}, \{6, 8\}\}$, $\mathcal{S}_5 = \{\{6, 7\}\}$, $\mathcal{S}_6 = \{\{5, 7\}, \{7, 8\}\}$, $\mathcal{S}_7 = \{\{5, 6\}, \{6, 8\}\}$. Since a Byzantine process can define its slices arbitrarily, it is not required to define its slices; however, correct processes must define their slices so that the maximal consensus cluster can emerge. Notice that, with these slices, there is a quorum for each correct process (e.g., 1's quorum is the area with horizontal lines, and 3's quorum is the area with vertical lines). Since all those quorums intersect at quorums of 5, 6, and 7 (i.e., $Q_5 = Q_6 = Q_7 = \{5, 6, 7\}$ — the area with squares), which are composed of correct processes, every two correct processes are intertwined. In this example, there are a few consensus clusters, such as $C_1 = \{5, 6, 7\}$ and $C_2 = \{1, 2, \ldots, 7\}$, but C_2 is the only maximal consensus cluster.

E. CUP Model

The Consensus with Unknown Participants (CUP) model [18] solves consensus in a distributed system where processes' knowledge about the system composition is incomplete. This model is useful for studying the necessary and sufficient knowledge conditions that processes require in order to solve consensus under different assumptions.

In CUP, the knowledge connectivity is encapsulated in an oracle called a *Participant Detector* (PD). A PD can be seen as a distributed oracle that provides each process hints about the participating processes in the distributed computation. Let PD_i be defined as the participant detector of a process i, such that PD_i returns a set of processes $\Pi_i \subseteq \Pi$ to which i can initially contact. We say a process j is a neighbor of another process i if and only if $j \in PD_i$. The information provided by the participant detectors of all processes forms a *knowledge connectivity graph* (see definition below), which is a directed graph since the initial knowledge provided by different PDs is not necessarily bidirectional, i.e., i knows j, but j might not know i.

Definition 5 (Knowledge connectivity graph [17])**.** Let $G_{di} = (V_{di}, E_{di})$ be the directed graph representing the knowledge relation determined by the PD oracle. Then, $V_{di} = \Pi$ and $(i, j) \in E_{di}$ if and only if $j \in PD_i$, i.e., i knows j.

It is important to remark that the knowledge connectivity graph defines the list of processes that every process initially knows in the system, *not their network's connectivity*. In CUP, at the beginning of the execution, each process i has only access to PD_i and f. Access to PD is required since processes cannot solve any nontrivial distributed coordination task without having some initial knowledge about other processes [17]. Furthermore, when n is unknown, processes cannot solve consensus in non-synchronous systems without knowing f [30].

The undirected graph obtained from the directed knowledge connectivity graph $G_{di} = (V_{di}, E_{di})$ is defined as $G = (V_{di}, \{(i, j) : (i, j) \in E_{di} \lor (j, i) \in E_{di}\})$. A component $G_{sink} = (V_{sink}, E_{sink})$ of G_{di} is a sink component if and only if there is no path from a node in G_{sink} to other nodes of G_{di}, except nodes in G_{sink} itself. A process $i \in V_{di}$ is a *sink member* if and only if $i \in V_{sink}$; otherwise, i is a *non-sink member*. See Fig. 1 for an example.

The Byzantine Fault-Tolerant (BFT) CUP problem can be solved under partial synchrony if the knowledge connectivity graph of processes satisfies the k-One Sink Reducibility property [17], [29]. This property ensures that every process can reach the sink members, and every correct sink member can discover the whole sink. As soon as the sink is discovered, sink members solve consensus among themselves by executing a consensus protocol (e.g., PBFT [3]). Then, they disseminate the decided value to non-sink members. Notice that having multiple sinks might violate the agreement property of consensus because each sink might remain unaware of other sinks until deciding some value, yielding the possibility of deciding distinct values.

Definition 6 ($k-$One Sink Reducibility (OSR) PD [17])**.** This class of PD contains all knowledge connectivity graphs G_{di} such that:

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

The *safe Byzantine failure pattern* defines the parameter k of k-OSR PD by considering the location of up to f failures in G_{di}.

Definition 7 (Safe Byzantine failure pattern [17])**.** Let G_{di} be a knowledge connectivity graph, f be the maximum number of processes in G_{di} that may fail, and F be the set of faulty processes in G_{di} during an execution. The safe Byzantine failure pattern for G_{di} and F is the graph $G_{safe} = G_{di} \setminus F$: $(F \subset G_{di}) \wedge (|F| \leq f) \wedge (G_{di} \setminus F \in (f+1)-\text{OSR})$.

If the safe Byzantine failure pattern holds during the execution of consensus in G_{di}, then we say that G_{di} is *Byzantine-safe* for F. The following theorem from [17] determines the minimal requirements to solve consensus in the BFT-CUP model.

Theorem 1. Consensus is solvable in the BFT-CUP model if there is a knowledge connectivity graph that is Byzantine-safe for F, and its sink component contains at least $2f+1$ correct processes.

Throughout the paper, when we use PD_i, where i is a process, we assume that the union of $PD_1, PD_2, \ldots, PD_{|\Pi|}$ forms a k-OSR graph that is Byzantine-safe for F and its sink component has at least $2f+1$ correct processes.

F. Threshold-based Analysis

Recall that both the Stellar and BFT-CUP models solve consensus under partial synchrony. Since it is proved that solving consensus without knowing n and f (or a fail-prone system in general) is impossible in non-synchronous systems [30], both Stellar and BFT-CUP require knowledge of f or a fail-prone system. However, only BFT-CUP explicitly considers f.

Since the main objective of this paper is to analyze the knowledge requirement of Stellar and compare it with the BFT-CUP model, we consider threshold-based systems for the Stellar model to have a common ground and also for simplicity, i.e., we use f to define slices and quorums. Consequently, in the remaining part of the paper, we say that a set I of correct processes is intertwined if any two quorums Q and Q' of members of I satisfy $|Q \cap Q'| > f$.

[1]A graph G is said to be k-strongly connected if, for any pair (i, j) of nodes in G, i can reach j through at least k node-disjoint paths in G.

IV. Defining Slices in the CUP Model

As mentioned previously, at the beginning of the execution, each process i has only access to PD_i and f in the CUP model. In the Stellar model, it has only access to its slices. This section focuses on the following question: *"Can slices be defined locally in the Stellar model using the information provided by the participant detectors to solve consensus in an unknown network with a known fault threshold?"* or, equivalently, *"Provided that each process i has only access to PD_i and f, can i define its slices locally to form intertwined quorums that lead to a maximal consensus cluster?"*

We negatively answer those questions by first presenting two necessary properties that must be satisfied by the slices defined by each process i using PD_i and f. Then, we show that two sets of processes Q_1 and Q_2 might be identified as quorums using `is_quorum` (Algorithm 1) without satisfying $|Q_1 \cap Q_2| > f$, i.e., it is impossible to ensure the formation of intertwined quorums if each process i defines its slices locally using just PD_i and f. In the following, we present such necessary properties as lemmas.

Lemma 1. Provided that the slices of each process i are defined locally using PD_i and f, every slice S of i is a subset of PD_i, i.e., $\forall i \in \Pi, \forall S \in \mathcal{S}_i : S \subseteq PD_i$.

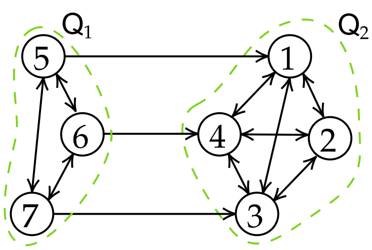
Proof: Initially, each process i only knows PD_i and f. Therefore, it can only define slices using processes contained in PD_i. ■

Lemma 2. Each correct process i must have at least one slice composed entirely of correct processes to solve consensus in Stellar. Formally, let $\mathcal{B}_i$ be equal to $\{\forall B \subset PD_i : |B| \leq f\}$, then $\forall B \in \mathcal{B}_i, \exists S \in \mathcal{S}_i : S \cap B = \emptyset$.

Proof: For the sake of contradiction, assume that i does not have any slice composed entirely of correct processes. That is, each slice of i has at least one faulty process. Since faulty processes can stay silent during an execution of a consensus instance, i might not be able to make progress. Therefore, i might not be able to solve consensus, which is a contradiction. ■

Theorem 2. If each process i defines its slices locally using PD_i and f, processes might violate the quorum intersection property.

Proof: We prove this theorem by showing a counter-example. Consider G_{di} as the one depicted in Fig. 2. This graph represents a 3-OSR PD (see Definition 6), with $V_{sink} = \{1, 2, 3, 4\}$, which provides enough knowledge for solving consensus with $f = 1$. Notice that whether the faulty process is a sink member or not, there are at least $2f + 1 = 3$ correct



Processes' knowledge:
$PD_1 = \{2, 3, 4\}$
$PD_2 = \{1, 3, 4\}$
$PD_3 = \{1, 2, 4\}$
$PD_4 = \{1, 2, 3\}$
$PD_5 = \{1, 6, 7\}$
$PD_6 = \{4, 5, 7\}$
$PD_7 = \{3, 5, 6\}$

Fig. 2. A knowledge connectivity graph satisfying 3-OSR PD definition. The dashed areas are two quorums, each formed by locally defined slices using PD and f.

sink members, there are at least $f+1 = 2$ node disjoint paths from any correct non-sink member to any correct sink member, and there are at least $f+1 = 2$ node disjoint paths from any correct sink member to another correct sink member.

We can define the set of slices of each process i as all subsets of PD_i with size $|PD_i| - 1$. In this way, we can ensure that each slice of i is a subset of PD_i (Lemma 1) and i has at least one slice composed entirely of correct processes (Lemma 2). Set $Q_1 = \{5, 6, 7\}$ is a quorum because every process $j \in Q_1$ has a slice inside Q_1. Likewise, $Q_2 = \{1, 2, 3, 4\}$ is also a quorum. Since $Q_1 \cap Q_2 = \emptyset$, the quorum intersection property is violated.

■

Corollary 1. Stellar cannot solve Byzantine consensus with the minimal knowledge connectivity requirement of consensus.

V. Defining Slices Using Sink Detector

We showed that Byzantine consensus might not be solved in the Stellar model if each process i locally defines its slices using PD_i and f. The major problem with this approach is that in k-OSR, sink members might form quorums that do not intersect quorums formed by non-sink members. In more detail, the knowledge connectivity graph obtained from PDs might be directed, and sink members do not know initially about non-sink members. Hence, a quorum formed by sink members contains only themselves. On the other hand, non-sink members can form quorums without including sink members (see Theorem 2).

This problem can be solved by making non-sink members include the sink members in their slices. Notice that the reverse is not possible since sink members do not have knowledge about non-sink members. This section introduces an oracle called *sink detector* through which processes can discover the members of the sink component of a k-OSR knowledge connectivity graph.

Definition 8 (Sink detector)**.** The Sink Detector (SD) is an oracle that provides an operation `get_sink`. Each process i must provide PD_i and f as input to `get_sink`, which satisfies the following properties:

- If $i \in V_{sink}$, it returns $\langle true, V \rangle$ to i, where $V = V_{sink}$, and
- If $i \notin V_{sink}$, it returns $\langle false, V \rangle$, where $V \subseteq V_{sink}$ contains at least $f+1$ correct members of V_{sink}.

Algorithm 2 Building slices – code of process i.

function `build_slices`(PD_i, f)
1: $\langle flag, V \rangle \leftarrow$ `get_sink`(PD_i, f)
2: **if** $flag = true$
3: $\quad \mathcal{S}_i \leftarrow$ all subsets of V with size $\lceil (|V| + f + 1)/2 \rceil$
4: **else**
5: $\quad \mathcal{S}_i \leftarrow$ all subsets of V with size $f+1$
6: **return** $\mathcal{S}_i$

It is important to remark that given a tuple $\langle *, V \rangle$ returned by `get_sink`, V might contain faulty processes.

Defining slices using SD. Each process i can get its slices by executing function `build_slices`, defined in Algorithm 2. This function defines slices using SD. In further detail, first, `build_slices` obtains the sink members by calling `get_sink` (line 1), which is based on PD_i and f. Then, it defines slices based on whether i is a sink member or not, as follows:

- If $i \in V_{sink}$, $\mathcal{S}_i$ contains all subsets of V with size $\lceil (|V| + f + 1)/2 \rceil$ (line 3).
- If $i \notin V_{sink}$, $\mathcal{S}_i$ contains all subsets of V with size $f+1$ (line 5).

The main idea behind defining slices this way is to enable us to define a lower bound for the size of quorums. Recall that a set Q is a quorum if each member of Q has a slice contained within Q. Since slices defined using Algorithm 2 differ based on whether a process is inside or outside of the sink, quorums of the sink members will be different from quorums of non-sink members:

- *Quorums formed by sink members:* Consider any correct process $i \in V_{sink}$ and its respective quorum Q_i. Since i's slices contain only sink members of size $\lceil (|V_{sink}| + f + 1)/2 \rceil$, Q_i contains only sink members. Furthermore, Q_i's size is greater than or equal to $\lceil (|V_{sink}| + f + 1)/2 \rceil$.
- *Quorums formed by non-sink members:* Consider any correct non-sink member j and its respective quorum Q_j. From the definition of quorums, process j must have a slice S contained within Q_j. Since each slice of j contains $f+1$ sink members (lines 4-5 of Algorithm 2), there is at least one correct sink member among them. Consequently, S must contain a correct sink member v. Since each slice of v has size $\lceil (|V_{sink}| + f + 1)/2 \rceil$ and Q_j must contain a slice of v, Q_j's size is greater than or equal to $\lceil (|V_{sink}| + f + 1)/2 \rceil$.

Correctness proofs. By defining slices using Algorithm 2, Stellar can solve consensus on the CUP model. We prove this result by showing that every two correct processes are intertwined. In further detail, we show how the sink and non-sink members are intertwined through three lemmas, as shown in Fig. 3. At a high level, those lemmas show that every two

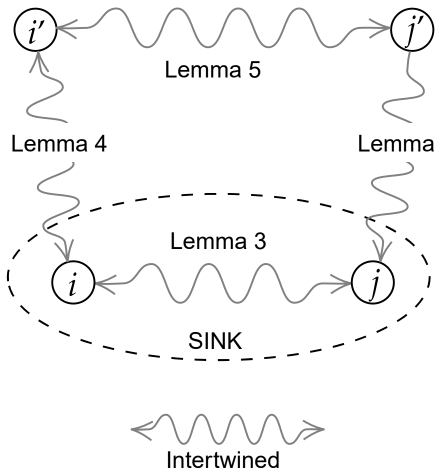


Fig. 3. Illustration of the approach used to prove that each correct process is intertwined with another correct process. Processes i, i', j, and j' are correct. Processes i and j are sink members.

correct processes' quorums intersect in at least $f+1$ sink members, i.e., every two correct processes are intertwined through the sink. We put together these three lemmas in a theorem, proving the result.

The following lemma shows that every two correct sink members are intertwined due to their quorum intersections.

Lemma 3. If slices of each sink member are defined using Algorithm 2, then any two correct sink members are intertwined.

Proof: Consider any two (possibly different) processes $i, j \in V_{sink}$. Let Q_i be any quorum of i and Q_j be any quorum of j. To prove the lemma, we need to show that $|Q_i \cap Q_j| > f$. First, notice that $Q_i, Q_j \subseteq V_{sink}$. Since the sizes of both Q_i and Q_j are greater than or equal to $\lceil (|V_{sink}|+f+1)/2 \rceil$, and there are at most f faulty processes, it follows that $|Q_i \cap Q_j| > f$. ■

The next lemma shows that every correct non-sink member is intertwined with every correct sink member because their quorums' intersections contain at least $f+1$ sink members.

Lemma 4. If slices of each process are defined using Algorithm 2, then any correct sink member and any correct non-sink member are intertwined.

Proof: Consider any correct non-sink member i' (resp. correct sink member i) and its quorum $Q_{i'}$ (resp. Q_i). We need to show that i' and i are intertwined. Recall that any slice of i' contains $f+1$ sink members, and according to Definition 1, $Q_{i'}$ contains slices of those sink members. Since any slice of each sink member has size $\lceil (|V_{sink}|+f+1)/2 \rceil$, $Q_{i'}$ contains at least $\lceil (|V_{sink}|+f+1)/2 \rceil$ members of the sink. On the other hand, quorum Q_i contains at least $\lceil (|V_{sink}|+f+1)/2 \rceil$ sink members, as i is a sink member, and each of its slices contains at least $\lceil (|V_{sink}|+f+1)/2 \rceil$ sink members. Therefore, $|Q_{i'} \cap Q_i| > f$, i.e., i' and i are intertwined. ■

It remains to show that any two correct non-sink members are also intertwined. The following lemma formalizes it by showing that any correct non-sink member is intertwined with another correct non-sink member through sink members.

Lemma 5. If slices of each process are defined using Algorithm 2, then any two correct non-sink members are intertwined.

Proof: Consider any two correct non-sink members i' and j'. Consider any quorum $Q_{i'}$ of i' and any quorum $Q_{j'}$ of j'. Let i and j be any two correct sink members such that $i \in Q_{i'}$ and $j \in Q_{j'}$. This is a valid assumption because quorums of non-sink members have at least $f+1$ sink members. Since every member of $Q_{i'}$ has a slice contained within $Q_{i'}$, $Q_{i'}$ is also a quorum of i. Similarly, $Q_{j'}$ is also a quorum of j. Due to Lemma 3, i and j are intertwined, i.e., $|Q_i \cap Q_j| > f$. Accordingly, any quorum of i' with any quorum of j' has an intersection containing at least $f+1$ sink members, meaning that i' and j' are intertwined. ■

Theorem 3. If slices of each process are defined using Algorithm 2, then any two correct processes are intertwined.

Proof: The proof follows directly from Lemmata 3, 4, and 5. ■

Recall that, from Definition 3, Stellar requires a single maximal consensus cluster to solve consensus. Theorem 3 shows that when processes use PD, f, and SD to build slices, every two correct processes will be intertwined, which is one of two properties of consensus cluster. The following theorem proves that PD, f, and SD are sufficient to ensure the second property of consensus cluster, i.e., each correct process has at least one quorum composed entirely of correct processes.

Theorem 4. Let G_{di} be a knowledge connectivity graph with a sink component containing at least $2f+1$ correct processes. If slices of each process in G_{di} are defined using Algorithm 2, then any correct process has at least one quorum composed entirely of correct processes.

Proof: Let $G_{sink} = (V_{sink}, E_{sink})$ be the sink component of G_{di}. We need to show that each correct process i has at least one quorum composed entirely of correct processes. We consider two cases:

1) $i \in V_{sink}$. Let F_{sink} be the set containing all faulty processes of the sink. Recall that each subset of V_{sink} with size $\lceil (|V_{sink}|+f+1)/2 \rceil$ is a slice of i. We first show that i has at least one slice composed entirely of correct processes, i.e., $\exists S \in \mathcal{S}_i$ such that $S \cap F_{sink} = \emptyset$. To do so, we need to show that the following inequality holds:

$$|V_{sink}| \geq |F_{sink}| + \lceil (|V_{sink}|+f+1)/2 \rceil \qquad (1)$$

Since $|V_{sink}|$ and $|F_{sink}|$ are two natural numbers, Inequality 1 holds if and only if the following inequality holds:

$$|V_{sink}| \geq |F_{sink}| + (|V_{sink}|+f+1)/2 \qquad (2)$$

After simplifying Inequality 2, we have:

$$|V_{sink}| \geq f + 1 + 2|F_{sink}| \quad (3)$$

Since $|F_{sink}| \leq f$, we have:

$$2f + 1 + |F_{sink}| \geq f + 1 + 2|F_{sink}| \quad (4)$$

Since the sink component of G_{di} contains at least $2f+1$ correct processes, we have:

$$|V_{sink}| \geq 2f + 1 + |F_{sink}| \quad (5)$$

By setting Inequalities 4 and 5 as the base and taking backward steps, it follows that Inequality 1 holds. Next, we show that a set $Q = S \cup \{i\}$ is a quorum for i. To do so, we need to show that $\forall j \in Q$, j has a slice in Q. Notice that Q is composed of correct processes and its size is $\lceil (|V_{sink}| + f + 1)/2 \rceil + 1$. Since any subset of V_{sink} with size $\lceil (|V_{sink}| + f + 1)/2 \rceil$ is a slice for j, j has a slice in Q.

2) $i \notin V_{sink}$. Due to Definition 8, i has at least one slice S' composed of correct sink members. From the first case, there must be a quorum Q' composed of correct processes that is a quorum for each member of S'. Notice that each member of $Q'' = Q' \cup \{i\}$ has a slice in Q'', so Q'' is a quorum for i.

The theorem holds since i has at least one quorum composed only by correct processes in both cases. ∎

Theorem 5. Let G_{di} be a knowledge connectivity graph that is Byzantine-safe for F, and its sink component, $G_{sink} = (V_{sink}, E_{sink})$, contains at least $2f+1$ correct processes. PD, f, and SD are sufficient to solve consensus in Stellar.

Proof: We need to show that all correct processes form only one maximal consensus cluster using PD, f, and SD. According to Theorem 3, every two correct processes are intertwined, which ensures the Quorum Intersection property. From Theorem 4, each correct process has a quorum composed entirely of correct processes, which ensures Quorum Availability. Since both properties of the consensus cluster are ensured, all correct processes form a consensus cluster C using PD, f, and SD. Since C contains all correct processes, it is maximal, proving the theorem. ∎

Corollary 2. Having access to a sink detector, Stellar can solve consensus with the minimal knowledge connectivity requirement of Byzantine consensus.

VI. Implementing the Sink Detector

This section presents an implementation of the sink detector using only the minimal knowledge required for solving consensus, i.e., the union of $PD_1, PD_2, \ldots, PD_{|\Pi|}$ forms a k-OSR graph that is Byzantine-safe for F, and its sink component has at least $2f+1$ correct processes. This oracle discovers and returns members of the sink component. When the `get_sink` function is called, there are two ways to discover the sink. The first way is to discover the sink directly. However, it might be impossible, requiring the indirect discovery of the sink. In the following, we elaborate on each way.

Discovering the sink directly. Each process i calls `SINK`(PD_i, f) function, presented in [17], to discover the sink directly. In a nutshell, `SINK` consists of three steps:

1) It runs a kind of breadth-first search in G_{di} to obtain the maximal set of processes that i can reach and stores it in a variable $known_i$. Every sink member terminates this step; however, a non-sink member might not be able to terminate (to see the reason, see Section 4 of [17]).
2) After obtaining $known_i$, process i sends $known_i$ to every process it knows.
3) If i receives at least $|known_i| - f$ messages with the same content as $known_i$, then i is a sink member, and the algorithm terminates by returning $\langle true, V_{sink} \rangle$. Otherwise, if i receives at least $f+1$ messages with different sets than $known_i$, it is a non-sink member.

The following lemma proves that `SINK` terminates at sink members by returning the sink members. See [17] for the proof.

Lemma 6 (Sink members – `SINK` [17])**.** Function `SINK` executed by a correct process $i \in G_{sink}$ satisfies the following properties:

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

Since non-sink members might not terminate the first step in the `SINK` function, they cannot use `SINK` to discover the sink directly. Hence, in addition to executing `SINK`, each process i might need to discover the sink indirectly.

Discovering the sink indirectly. Each process i asks sink members to send the sink component to it. If a sink member discovers the sink and receives i's request, it sends the sink to i. A primitive called *reachable-reliable broadcast*, also presented in [17], is used by i to communicate with sink members. The primitive provides two operations:

- `reachable_bcast`(m, i) – through which the process i broadcasts message m to all f-reachable processes from i in G_{di}.
- `reachable_deliver`(m, i) – invoked by a receiver to deliver message m sent by the process i.

This primitive is based on the notion of f-reachability.

Definition 9 (f-reachability [17])**.** Consider a knowledge connectivity graph G_{di} and let f be the number of processes in G_{di} that may fail. For any two processes $i, j \in G_{di}$, j is f-reachable from i in G_{di} if there are at least $f+1$ node-disjoint paths from i to j in G_{di} composed only by correct processes.

The reachable-reliable broadcast should satisfy the following properties:

- *RB_Validity*: If a correct process i invokes `reachable_bcast`(m, i) then (i) some correct process

j, f-reachable from i in G_{di}, eventually invokes `reachable_deliver`(m, i) or (ii) there is no correct process f-reachable from i in G_{di}.

- *RB_Integrity*: For any message m, if a correct process j invokes `reachable_deliver`(m, i) then process i has invoked `reachable_bcast`(m, i).
- *RB_Agreement*: If a correct process j invokes `reachable_deliver`(m, i), where m was sent by a correct process i that invoked `reachable_bcast`(m, i), then all correct processes f-reachable from i in G_{di} invoke `reachable_deliver`(m, i).

This primitive was implemented in asynchronous systems, and it was shown that all sink members are f-reachable from any process in G_{di} [17]. Therefore, if any process broadcasts a message using `reachable_bcast`, all correct sink members deliver the message using `reachable_deliver`. Accordingly, any non-sink member will discover the sink members with the help of sink members.

Description of `get_sink` (Algorithm 3). When `get_sink` is called, each process i examines whether it has discovered the sink. If it is not the case, it asks processes to send the sink to it by broadcasting a message tagged with `GET_SINK` (line 5). By delivering a message tagged with `GET_SINK` sent by a process j, i adds j to the set *asked* (line 17). Then, i executes `SINK`(PD_i, f). If $i \in V_{sink}$, `SINK` terminates by returning $\langle true, V_{sink} \rangle$, and i sends V_{sink} to every member of *asked* (lines 18-21). Otherwise, i must wait until the sink members send the sink to it. By receiving a value v from any other process, i adds v to the list *values*. If there is a value v that is repeated more than f times in *values*, i selects v as the sink (lines 15-16). As soon as i finds the sink, it returns the sink.

Theorem 6. If a correct process calls `get_sink` (Algorithm 3), it will eventually receive V_{sink}.

Proof: Let i be a correct process. We need to consider two cases:

1) $i \in V_{sink}$. Since the invocation of `SINK` terminates by returning $\langle true, V_{sink} \rangle$ to i according to Lemma 6, the theorem holds for this case.
2) $i \notin V_{sink}$. Notice that members of the sink are f-reachable from i, so every correct sink member will receive $\langle$`GET_SINK`$, i\rangle$. Since `SINK` terminates in every correct process $j \in V_{sink}$, j will obtain V_{sink} and can send it to i. Since there are at least $2f + 1$ correct processes inside the sink, i will receive more than f values that are equal to V_{sink}. It follows that i can eventually learn V_{sink}.

■

Each process uses Algorithm 3 to obtain the sink members, which are used in Algorithm 2 to define its slices forming a consensus cluster.

Algorithm 3 SD code of process i.

---

variable
1: *sink* $\leftarrow \emptyset$ /∗ a set that will be filled with V_{sink} eventually ∗/
2: *asked* $\leftarrow \emptyset$ /∗ a set containing the ids of processes that asked i about the sink ∗/
3: *values* $\leftarrow \emptyset$ /∗ a list containing the values returned by other processes ∗/

function `get_sink`(PD_i, f)
4: **if** *sink* $= \emptyset$
5: `reachable_bcast`$($`GET_SINK`$, i)$
6: **fork** `wait_sink()`
7: **if** $\langle true, V_{sink}\rangle =$ `SINK`(PD_i, f) /∗ executing the `SINK` algorithm from [17] ∗/
8: *sink* $\leftarrow V_{sink}$
9: **fork** `send_sink()`
10: **wait until** *sink* $\neq \emptyset$
11: **if** $i \in$ *sink*
12: **return** $\langle true, sink\rangle$
13: **else**
14: **return** $\langle false, sink\rangle$

function `wait_sink()`
15: **wait until** there is a value v that is repeated more than f times in *values*
16: *sink* $= v$

upon `reachable_deliver`$($`GET_SINK`$, j)$
17: *asked* $\leftarrow$ *asked* $\cup \{j\}$

function `send_sink()`
18: **loop**
19: **if** there is a process $j \in$ *asked*
20: **send** $\langle$`SINK`$, sink\rangle$ to j
21: *asked* $\leftarrow$ *asked* $\setminus \{j\}$

upon receiving $\langle$`SINK`$, V\rangle$
22: *values* $\leftarrow$ *values* $\cup \{V\}$

VII. Conclusion

We studied the required knowledge for Stellar to solve consensus in open systems. We showed that it is impossible to ensure the formation of a consensus cluster when each participant defines its slices locally using the fault threshold and a list of participants defined by the minimal knowledge connectivity graph required for solving Byzantine consensus. We also proposed an oracle – the sink detector – that provides the information required by each participant to define slices that lead to the formation of a consensus cluster.

These results imply that, differently from the BFT-CUP protocol [17], Stellar cannot solve consensus when processes have only the minimal required knowledge about the system. Further, to make Stellar solve consensus in such conditions, processes need to run some distributed knowledge-increasing protocol before building their slices. An interesting question for future work is if the BFT-CUP approach can be used for implementing a permissionless blockchain.

ACKNOWLEDGMENTS

We thank the ICDCS'23 anonymous reviewers for providing constructive comments to improve this paper. This work was supported by FCT through a Ph.D. scholarship (2020.04412.BD), the SMaRtChain project (2022.08431.PTDC), and the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and by European Commission through the VEDLIoT project (H2020 957197).

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, 1982.

[2] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.

[3] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Symposium on Operating Systems Design and Implementation*, 1999.

[4] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, vol. 35, no. 2, 1988.

[5] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, 1998.

[6] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX Annual Technical Conference*, 2014.

[7] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[8] M. Vukolić, "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication," in *International Workshop on Open Problems in Network Security*, 2015.

[9] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus," in *International Conference on Principles of Distributed Systems*, 2018.

[10] C. Decker, J. Seidel, and R. Wattenhofer, "Bitcoin Meets Strong Consistency," in *International Conference on Distributed Computing and Networking*, 2016.

[11] R. Pass and E. Shi, "Hybrid Consensus: Efficient Consensus in the Permissionless Model," in *International Symposium on Distributed Computing*, 2017.

[12] Á. García-Pérez and A. Gotsman, "Federated Byzantine Quorum Systems," in *International Conference on Principles of Distributed Systems*, 2018.

[13] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and Secure Global Payments with Stellar," in *ACM Symposium on Operating Systems Principles*, 2019.

[14] D. Schwartz, N. Youngs, A. Britto *et al.*, "The Ripple Protocol Consensus Algorithm," https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

[15] D. Mazieres, "The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus," https://stellar.org/papers/stellar-consensus-protocol.pdf, 2015.

[16] G. Losa, E. Gafni, and D. Mazières, "Stellar Consensus by Instantiation," in *International Symposium on Distributed Computing*, 2019.

[17] E. A. P. Alchieri, A. Bessani, F. Greve, and J. da Silva Fraga, "Knowledge Connectivity Requirements for Solving Byzantine Consensus with Unknown Participants," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 2, 2018.

[18] D. Cavin, Y. Sasson, and A. Schiper, "Consensus with Unknown Participants or Fundamental Self-Organization," in *International Conference on Ad-Hoc Networks and Wireless*, 2004.

[19] ——, "Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes," EPFL - LSR, Tech. Rep., 2005.

[20] F. Greve and S. Tixeuil, "Knowledge Connectivity vs. Synchrony Requirements for Fault-Tolerant Agreement in Unknown Networks," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.

[21] I. Damgård, Y. Desmedt, M. Fitzi, and J. B. Nielsen, "Secure Protocols with Asymmetric Trust," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2007.

[22] B. Chase and E. MacBrough, "Analysis of the XRP ledger consensus protocol," *arXiv preprint arXiv:1802.07242*, 2018.

[23] I. Amores-Sesar, C. Cachin, and J. Mićić, "Security analysis of ripple consensus," in *International Conference On Principles Of Distributed Systems*, 2020.

[24] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing*, vol. 11, no. 4, 1998.

[25] Álvaro García-Pérez and M. A. Schett, "Deconstructing Stellar Consensus," in *International Conference On Principles Of Distributed Systems*, 2020.

[26] C. Cachin and B. Tackmann, "Asymmetric distributed trust," in *International Conference On Principles Of Distributed Systems*, 2019.

[27] C. Cachin, G. Losa, and L. Zanolini, "Quorum Systems in Permissionless Network," in *International Conference On Principles Of Distributed Systems*, 2022.

[28] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, 1996.

[29] E. A. Alchieri, A. N. Bessani, J. d. Silva Fraga, and F. Greve, "Byzantine Consensus with Unknown Participants," in *International Conference On Principles Of Distributed Systems*, 2008.

[30] P. Khanchandani and R. Wattenhofer, "Byzantine Agreement with Unknown Participants and Failures," in *2021 IEEE International Parallel and Distributed Processing Symposium*, 2021.

[31] A. Momose and L. Ren, "Constant Latency in Sleepy Consensus," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[32] R. Pass and E. Shi, "The Sleepy Model of Consensus," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[33] M. Biely, P. Robinson, and U. Schmid, "Agreement in Directed Dynamic Networks," in *International Colloquium on Structural Information and Communication Complexity*, 2012.

[34] N. H. Vaidya, L. Tseng, and G. Liang, "Iterative Approximate Byzantine Consensus in Arbitrary Directed Graphs," in *ACM Symposium on Principles of Distributed Computing*, 2012.

[35] M. Biely, P. Robinson, U. Schmid, M. Schwarz, and K. Winkler, "Gracefully Degrading Consensus and k-set Agreement in Directed Dynamic Networks," *Theoretical Computer Science*, vol. 726, 2018.

[36] L. Tseng and N. H. Vaidya, "Fault-Tolerant Consensus in Directed Graphs," in *ACM Symposium on Principles of Distributed Computing*, 2015.

[37] J. R. Douceur, "The Sybil Attack," in *International Workshop on Peer-to-Peer Systems*, 2002.

[38] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *International Conference on Fundamentals of Computation Theory*, 1983.

[39] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," in *Annual Symposium on Theory of Computing*, 1997.