HAL open science

Abstractions to Build Permissionless Systems

Emmanuelle Anceaume

▶ To cite this version:

Emmanuelle Anceaume. Abstractions to Build Permissionless Systems. Computer Science [cs]. Université de Rennes 1, 2019. tel-04313136

HAL Id: tel-04313136
https://hal.science/tel-04313136v1

Submitted on 29 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HABILITATION À DIRIGER LES RECHERCHES

présentée devant

L'Université de Rennes 1
Spécialité: Informatique

par

Emmanuelle Anceaume

Abstractions to Build Permissionless Systems

Defended on the 18-th of December, 2019

devant le jury composé de:

Amr El Abbadi Antonio Fernández Anta Nicolas Hanusse
Fernando Pedone Yvonne-Anne Pignolet Francois Taiani
Philippas Tzigas

et au vu des rapports de:

Amr El Abbadi Nicolas Hanusse Yvonne-Anne Pignolet

À Lise, Clémentine, Vincent et Laurent

Chapter 1

Dependable Distributed Computing

1.1 Introduction

Most of our current activities highly depend on distributed systems and applications. A distributed system is a collection of sequential computing entities (i.e., processes, processors, nodes, agents, peers) that communicate with one another by exchanging messages.

Distributed computing is the field that aims at coordinating distributed entities to solve a common non-trivial task. Fault tolerant distributed computing is the one that aims at making entities cooperate to reach a common goal in presence of failures. Solving and analyzing fault tolerant distributed computing problems is a notoriously difficult endeavor. First, and for many reasons, communications among entities or the rate at which computing entities execute can be arbitrarily delayed or simply cannot be bounded. Such an *asynchrony* leads to the incapacity for the entities to determine the time at which events take place. Second, computing entities have solely access to the information they receive. They thus have a *local view* of the current state of the system and so are *uncertain* about the views of others. Finally, despite the fact that entities communicate with one another, unpredictable *failures* affect parts of the system. An entity under the control of an adversary may misbehave deliberately in order to prevent communication among the remaining entities. Still, entities must *locally* take decisions or compute results whose effect is *global*. In other words, entities must have enough consistent information so that they can cooperate correctly and solve common non-trivial tasks.

While all distributed systems are concerned with unpredictability due to failures and synchrony issues, the particular attributes of large-scale and open systems, namely openness, extreme dynamism in terms of structure, content, and load, and lack of identities increase considerably uncertainty in the way entities cooperate. For instance, in peer-to-peer systems, nodes continuously join and leave the system, while in sensor, ad-hoc or robot networks, the energy fluctuation of batteries and the inherent mobility of nodes induce a dynamic aspect of the system. In all these systems there is no central entity in charge of entities organization and control, and there is an equal capability, and responsibility entrusted to each of them to participate to the execution of common tasks. To cope with such characteristics, these systems must be able to *self-organize*, i.e., to spontaneously organize toward desirable global properties. For instance, in peer-to-peer systems, self-organization is handled through protocols for node arrival and departure.

As it is the case for static distributed systems, large-scale and open systems must be based on a theoretical model, able to encapsulate the dynamic behavior of these systems. Reducing the specification of self-organization to the behavior of the system during its non-dynamic periods is not satisfying essentially because these periods may be very short, and rare. On the other hand, defining self-organization as a simple convergence process towards a stable predefined set of admissible

configurations is inadequate since due to the dynamic behavior of nodes, it may happen that no execution of the system converges to one of the predefined admissible configurations. Apprehending such dynamic systems goes through an analysis of the principles that govern them, namely high interaction, dynamics, and heterogeneity. High interaction relates to the propensity of nodes to continuously exchange information or resources with other nodes around them. The dynamics of these systems refer to the capability of nodes to continuously move around, to join or to leave the system as often as they wish based on their local knowledge. Finally heterogeneity refers to the specificity of each entity of the system: some have huge computation resources, or have large memory space, or are highly dynamic. In contrast, seeing such systems as a simple mass of entities completely obviates the differences that may exist between individual entities; those very differences make the richness of these systems.

Still due to the uncertainty created by the environment, distributed computing is characterized by a large number of negative results, either in terms of pessimistic lower bounds or impossibility results. Clearly these negative results are often obtained when considering general abstract models, however they are very important to determine when it is useless to try to design a solution for a given problem assuming a given model of execution. Fortunately, as emphasized by Attiya and Welch [55], distributed computing offers lot of room to maneuver: You can slightly weaken the problem statement by for example playing on the termination property; or you can make stronger assumptions on synchrony, faulty behaviors, failure detection capacity, or locality scope. Quoting Fisher and Meritt, "[...] by continuously revisiting the underlying assumptions with the goal of relaxing too restrictive ones, by searching for the right abstractions to capture the essence of real-world problems, and by finding the right primitives to be composed, distributed computing should be able to provide designers with a non distributed view of a distributed systems" [121].

In distributed computing, an algorithm is run by a set of entities (whose membership or cardinality are not necessarily known). Each entity, based on its input value or local state, cooperates with the other entities to compute its own output value. Despite the fact that distributed algorithms are often made of a few lines, the uncertainty created by the environment may make their behavior difficult to understand. Starting from the same initial state, different (correct) executions are possible. Thus, understanding the behaviour of a distributed algorithm means stating some essential *properties* that must hold for any execution of the algorithm under concern [162]. There are two kinds of properties a distributed algorithm should satisfy. *Safety* properties which say that some particular "bad" thing never happens, that is, that executions never enter an unacceptable state and, *liveness* properties which say that some particular "good" thing eventually happens, that is, that executions eventually enter a desirable state [7, 162]. Alpern and Schneider [7] showed that any property of a distributed system execution can be viewed as a composition of a liveness and a safety property, both properties justifying the confidence that can be placed in the algorithms [55].

Permissionless vs Permissioned Systems In this manuscript, I will extend the *permissioned* and *permissionless* terminology, which initially appeared with the blockchain technology, to distinguish between traditional distributed systems and large-scale and open ones. More precisely, by *permissionless* systems I mean distributed systems in which *(i)* the number of participants for carrying out the protocol is not known before hand, and is not even known during the course of the execution, *(ii)* the right to contribute (e.g., by proposing a new value of interest for the other participants) and to participate (e.g., by reading new decided values) is not controlled by a (trust-

worthy) third authority, and *(iii)* participants communicate over a weakly connected communication topology (such as a peer-to-peer network). In contrast, *permissioned* systems represent distributed systems in which *(i)* the set of participating entities for carrying out the protocol is fixed and known before hand, *(ii)* participants are authorized nodes, and *(iii)* participants communicate over a completely connected point-to-point communication topology.

The ability of permissionless systems to self-adapt to changes does not exist in permissioned systems. The reason is that any change in permissioned systems needs to reach a common explicit agreement. In permissionless systems this ability guarantees that even in case of unpredictable multiple changes, updates of local data structures, such as routing tables, are achieved independently by each entity without any risk of deadlocks and without requiring neither an explicit agreement, nor the deployment of a predefined configuration of the system. As will be shown along this manuscrit, such a radical gap on the scale and dynamicity of systems militates in favor of a paradigm shift for designing solutions to the problems raised by permissionless systems.

In this "research direction habilitation (HdR)" manuscript I will synthesize some of the works we have jointly achieved with my colleagues (master students, PhD students, and researchers) to get a better understanding of fundamental abstractions that help us to build both permissioned and permissionless fault tolerant distributed systems.

1.2 Models for Message-Passing Systems

1.2.1 Processes and Communication Graph

This section presents the abstractions we use to model a distributed system composed of processes that perform computations and cooperate by sending and receiving messages using a communication network. In the following, a process will abstract any entity that does computations, while the communication network will abstract the physical and logical network that allows any two processes to communicate among each other.

The physical network is described either by a strongly connected graph (in case of permissioned systems), or by a weakly connected graph (in case of permissionless systems). Its vertices represent processes of the system and its edges represent communication links between processes. Compared to permissioned systems, in permissionless ones, the physical communication graph is subject to frequent and unpredictable changes that are mainly caused by volunteer actions, such as joins and leaves of processes in P2P systems.

On top of the physical communication graph, which represents the possible communication links between processes within the system, we may define *logical layers*. A logical layer is dependent on the semantics of the running application, and thus on the data this application manipulates. For instance in publish/subscribe applications, the logical layer formed by the processes to publish data and to subscribe to the set of data they are interested in, may resemble a forest of trees. A logical layer is thus a weakly connected graph, where the vertices of the graph are the processes and the links of the graph are the logical links between these processes. Processes may belong to several layers simultaneously, and thus may have different sets of neighbors, one per each logical

layer. A logical layer is often referred to as the logical structured overlay when the logical graph is structured, or to the logical unstructured overlay when the graph is random.

1.2.2 Failure models and Adversaries

Understanding and controlling the system behavior in the presence of failures is an important endeavor. In order to make a distributed system fault tolerant, whatever its size or dynamism, one must be able to model the types of failures that may occur, and which component may be impacted (processes and communication network). Classically failure are modeled as benign or malign ones.

Benign failures characterize failures that unexpectedly prevent, permanently or temporarily, processes to execute their code, or the communication network to propagate information to their recipient. Citing Laprie [157], "[..] *benign failures are those whose consequences are of the same order of magnitude as those of the service delivered in the absence of failures*". This typically includes crash, omission and eavesdropping failures. In general we assume that a crashed process stop executing its local code and does not send any message. A process omission is more general, and is often caused by a buffer overflow. An eavesdropping failure of a process characterizes the fact that this process leaks information to non authorized processes. Concerning the communication network, a crash failure or a partitioning represents the impossibility for a subset of the processes to communicate with the rest of the system; Omission failures are due to network congestion that cause messages to be lost; and eavesdropping failures represent the reading by non authorized processes of transmitted messages.

On the other hand, *malign failures* "[...] *are those whose consequences are not commensurable with those of the service delivered in the absence of failures*" [157]. This captures all failures, ranging from accidental and erratic memory bit flips to malicious attacks. Accidental failures are assumed to occur independently and are often modeled as a random process. Malicious failures, also called Byzantine failures, are intentional and often result from a well-defined strategy orchestrated by an adversary which coordinates the actions of manipulated processes. While considering the potential occurrence of Byzantine failures make the design of distributed algorithms complex and difficult to prove, the attraction for such a model comes from the fact that we do not have to precisely characterize the type of behavior a faulty entity will exhibit, since everything is permitted from a Byzantine entity. In other words, the assumption coverage [181] of the Byzantine failure mode is 1 (the assumption coverage refers to the probability that the assumptions about the failure modes or the underlying network hold). A process that does not fail is often called *correct* or *honest*.

From a permissionless distributed computing perspective, processes may, in addition to these behaviors, exhibit a rational behavior. They are rational because they wish to maximize some utility function, and thus choose the appropriate actions to do so. As emphasized by Shneidman and al. [195], different approaches may be adopted to face rationality: *i)* to ignore it and to expect that the system will do its best despite self-interested processes, *ii)* to limit the effect that a rational process can have on the system by using trusted mechanisms, or *iii)* to adopt the fault tolerance techniques. Clearly none of these approaches benefits from resources that may be potentially offered by these rational processes (e.g., storage, computation, bandwidth). Thus the designed algorithm should incentive each process to behave rationally in a system efficient way. By following the same

characterization as the one advocated by Laprie, we might classify rational strategies as "benign" ones if their consequences are of the same order of magnitude as those of the service delivered in the absence of such strategies. For instance, rational processes may follow a *free-riding* strategy to minimize their participation, or they may follow a *greedy* strategy to maximize their access to services [28]. It is important to remark that in some specific contexts, processes may show irrational strategies. A typical example of irrational strategies has been observed in the early 2000s in peer-to-peer file sharing applications by the presence of peers that continue to share music even if there are disincentives for sharing (e.g., lawsuits). Similarly, in permissionless blockchains, processes forward blocks and transactions with no straightforward incentive other than making this technology sustainable in the long term. When considering systems that are expected to be sustainable in the long term, the online characterization of benign failures may also be important to rapidly determine the extent of their damage [17, 34, 38]. On the other hand, one may look at *Sybil attacks*, or *eclipse attacks* as malign strategies. A *Sybil attack* results from the generation of numerous fake identities to simulate the presence of many processes [104]; an *eclipse attacks* (also called *routing table poisoning*) represent the manipulation of the routing tables of honest processes (in the objective of having honest processes redirect outgoing links towards non legitimate ones, "eclipsing" honest processes from each others' view), or the dropping or re-routing of messages towards non legitimate recipients [77, 199].

To study the impact of failures, an execution of a distributed algorithm is often seen as a game between the algorithm and an adversary that tries to play with failures and with the scheduling of messages at the most strategic time to cause the algorithm to fail. An algorithm fails when its safety or liveness properties do not hold. The adversary wins the game if it is capable to make the algorithm fail, while the algorithm wins if it is tolerant to the adversarial strategies. For problems to be solvable, the adversary must be limited. The adversary is often limited *(i)* on the number of failures and the number of faulty entities it may choose during any execution of the algorithm; *(ii)* on the amount of information it knows prior or during the execution. For instance for randomized algorithms, the adversary may fix the scheduling and failure pattern independently of the random choices made by the randomized algorithm, or observes the results of the local coin flips, together with the state of all computing entities before scheduling the next entity to make a step; and finally *(iii)* on the computational power with which the adversary chooses its actions. The common assumption is the existence of a polynomially bounded adversary, that is an adversary which cannot in a polynomial number of steps or time or space break any abstraction of the protocol. Another way to limit the power of the adversary is to consider the time it takes for the adversary to corrupt entities, after it has decided who to corrupt. If the adversary is capable to instantaneously corrupt entities, the adversary is called *strongly adaptive* or *rushing* adversary. If on the other hand, it takes some time for the adversary to corrupt the entities it has chosen then, the adversary is called *weakly adaptive*.

1.2.3 Timing Assumptions

The assumptions regarding the passage of time is the other important constituent of distributed system models. Most of the distributed models in order to remain tractable have considered the following three timing models: *Synchronous, asynchronous* and *partially synchronous* models. In the synchronous model, a fixed upper bound Δ on the time it takes for a message to be trans-

mitted between any two processes, and a fixed upper bound Φ on the time that elapses between to consecutive steps of a process exist and are known. This model allows us to organize the execution of a distributed algorithm in rounds, such that in each round, every process can send a message to each of its neighbors, can receive the messages sent during the round, and can execute a computational step based on the received messages. Such a model allows us to design algorithms in which uncertainties are limited to the ones caused by failures [162]. Furthermore impossibility results for the synchronous model apply to less constrained models. The asynchronous model does not assume that both Δ and Φ exist, that is processes may take steps in an arbitrary order, at arbitrary relative speeds, and messages do not have to be delivered in the order they were sent. Asynchronous algorithms are designed independently of any particular timing assumptions, which makes them general, and thus can be run in any distributed system. On the other hand, weakness of this model often leads to negative results, either in terms of pessimistic lower bounds or impossibility results. Actually, asynchrony is excessively general for real distributed systems, but how to capture the amount of variability in real systems remains unclear [121]. For instance the consensus problem[1] is impossible to deterministically solve in the asynchronous model assuming that at most one process may crash, while it is deterministically solvable in a partially synchronous model assuming processes exhibit Byzantine faults. Partial synchrony lies between the synchronous model and the asynchronous one, and led to numerous timing models [102, 106], depending on whether both Δ and Φ exist but are not known, or whether bounds are known but they are only guarantee to hold from some unknown time T. In the former case this allows us to design algorithms that are correct regardless of the actual values of Δ and Φ, while in the later case, algorithms must be correct whatever the time at which T occurs. The relevance of the partially synchronous model is that it models the expected behavior of best-effort networks such as the Internet. Different notions of partial synchrony have been proposed, in particular the Θ model proposed by Widder and Schmid [209]. This model augments the asynchronous model by a (possibly unknown) bound Θ on the ratio of longest and shortest end-to-end delays of messages simultaneously in transit. Actually, an upper bound on those delays need not exist, however, and even Θ may hold only after some unknown global stabilization time.

1.2.4 Encapsulating Synchrony in an Abstract Entity

A powerful way to encapsulate synchrony is provided by the *failure detector abstraction* [82]. Failure detectors were introduced by Chandra and Toueg as a way to deterministically solve consensus in an asynchronous environment. The failure detector abstraction, also called failure detector oracle, provides information on crashed processes. It is defined in terms of axiomatic properties which states the minimal requirements on the detection of faulty entities without any explicit reference to physical time. *Completeness* properties provide information on the eventual detection of faulty processes, while *accuracy* properties provide information about which are correct. Both properties can be relaxed, giving rise to a combination of failure detectors ranging from *perfect failure detectors*, $\mathcal{P}$, which make no wrong suspicions and eventually detect every crash to *eventually weak failure detectors*, $\Diamond\mathcal{W}$, that guarantee that eventually there is a correct process which all processes will not falsely suspect and for every crash, eventually there is a process that will detect this crash. This abstraction has been the source of an extraordinary enthusiasm because it allows us to separate

[1] The consensus problem assumes that every entity initially proposes some value, and requires that entities eventually choose a common value among the proposed ones.

the concerns of timeout-based reasoning and the detection of failures, by encapsulating synchrony assumptions in an asynchronous interface. Under the argument that distinguishing synchrony and faults, or distinguishing process failures and link failures to solve consensus may lead to unnecessary intricate solutions and complex correctness analysis, Charron-Bost et al [84] have proposed the *Heard-of model*. This abstract structure encapsulates both synchrony and faults in the same abstract structure which allows us to handle any benign fault, be it static or dynamic, permanent or transient, in a unified framework [85]. Coming back to failure detectors, many others have been defined for solving various agreement problems. This includes, the failure detector Ω which eventually outputs the identity of a correct process, called leader, that is trusted by every other processes. Ω has been shown [88] to be equivalent to the eventually strong $\Diamond\mathcal{S}$ failure detector, which is the weakest failure detector to solve consensus [83]; The *quorum* failure detector Σ outputs at each process a set of processes such that any two sets intersect, and eventually every set output at correct processes consists only of correct processes. It was shown in [98] that Σ is the weakest failure detector to implement atomic registers. The *Heartbeat* failure detector introduced by Aguilera [132] can be used in quiescent protocols, that it protocols in which eventually processes do not send messages ; The *anonymously perfect* failure detector $?\mathcal{P}$ [131] also called *failure signal* failure detector $\mathcal{F}$ [98] can be used for solving the non-blocking atomic commit (NBAC) problem [66, 200]. With Roy Friedman and Maria Potop Butucaru, we have proposed a parametrized family of failure detectors $\mathcal{F}(A)$ such that $\mathcal{F}(A)$ is the weakest failure detector that enables solving the *managed agreement* problem given a set of specific processes, that we called aristocrats, A. The managed agreement problem generalizes both the problems of consensus and the non-blocking atomic commit [27]. Specifically, the definition of managed agreement is based on the notion of aristocrat processes, such that if any of the aristocrats proposes an input value, then the corresponding decision must be taken. On the other hand, if none of the aristocrats proposed an input value and none of the aristocrats failed, then any possible decision value that corresponds to a value that was proposed can be decided on. Thus, non-blocking atomic commit (NBAC) is a special case of managed agreement when all processes are aristocrats, whereas consensus is a special case of managed agreement when there are no aristocrats. We have proposed a generic algorithm for solving managed agreement, which is based on a transformation from consensus to NBAC by Guerraoui [131]. Our protocol utilizes any known consensus protocol as a black-box and a new class of failure detector that we denote $?\mathcal{P}_{\mathrm{Ar}}(A)$, which is an extension of $?\mathcal{P}$ to detect crashes of aristocrats only. Finally, we have introduced a failure detector class $\Psi_{\mathrm{Ar}}(A)$, again, an extension of the known class Ψ [98], and have shown that a corresponding family of failure detectors, denoted $\mathcal{F}(A) = (?\mathcal{P}_{\mathrm{Ar}}(A), \Psi_{\mathrm{Ar}}(A))$, is the weakest class of failure detectors that solves managed agreement for a given set of aristocrats A.

The Unreliable Distributed Timing Scrutinizer Another way to encapsulate synchrony is provided by the *Unreliable Distributed Timing Scrutinizer (UDTS)*. With Eric Mourgaya we have introduced the UDTS to capture the state of the network and based on its observation, this abstract structure estimates the waiting time that maximizes the efficiency of the protocol regarding the number and/or the duration of its computational rounds [40]. For example, maximizing the ratio of "the number of possible expected messages" to "the time needed to receive these messages" is a possible criterion to minimize the number and the duration of computational rounds. An Unreliable Distributed Timing Scrutinizer ($UDTS$) is completely defined by a completeness property, which essentially says that all stable links are detected. A communication link is stable if it satisfies the *locality* and *covering* properties. The *covering* property guarantees that the behavior of message

transmission delays over the communication link can always be approximated by a sequence of probabilistic laws, while the *locality* property guarantees that any two consecutive elements of this sequence are similar. The $UDTS$ oracle is unreliable in the sense that its estimations can be temporarily under-estimated or over-estimated, however the returned estimation is infinitely often correct. The UDTS mechanism has been also used to solve consensus [109], implement Chandra-Toueg's failure detectors [170], to evaluate the condition-based approach [169], and as an abstraction of the dynamic systems model proposed by Mostefaoui et al. [172].

1.2.5 Churn Model

As previously described, in permissionless systems the logical communication graph is subject to frequent changes. Whatever their causes, these changes are referred to as the *churn* of the system. Godfrey et al [130] define the churn as the sum, over each configuration, of the fraction of the system that has changed during that configuration, normalized by the duration of the execution. With Xavier Défago, Maria Potop Butucaru and Matthieu Roy we have proposed four classes of churn [22, 23].

- *Bounded dynamic churn model.* In this model, the number of connection and disconnection actions during any execution of the system is finite and a bound on this number is known. Note that in [171], the authors consider an α-parameterized system where the α parameter is a known lower bound on the number of processes that eventually remain connected within the system. In this later model augmented with some communication synchrony, the authors propose the implementation of the Ω failure detector oracle.

- *Finite dynamic churn model.* In this model, the number of connection and disconnection actions during any execution of the system is finite but the bound is unknown. That is, algorithms designed in this model cannot use the bound on the number of processes active in the system. Different forms of the finite dynamic churn model have been proposed in Aguilera [5], specifically the "finite arrival model (the *M2* scheduler) defined as a model allowing "infinitely many processes, but each run has only finitely many processes", and the "infinite arrival model" (the *M3* scheduler), which allows "infinitely many processes, runs can have infinitely many processes but in each finite time interval only finitely many processes take steps". Abboud et al [1] propose agreement algorithms compliant with this churn model.

- *Kernel-based dynamic churn model.* In this model, between any two successive static fragments of any execution of the system, there exists a non-empty group of processes for which the local knowledge is not impacted by connection and disconnection actions. Baldoni et al [61] extend this characterization to specific topological properties of the communication graph. Specifically, the authors define the churn impact with respect to the diameter of the graph, and distinguish three classes of churn. In the first one, the diameter of the graph is constant and every active node in the system can use this knowledge. In the second class, the diameter is upper bounded but active nodes do not have access to that upper bound. Finally, in the third one the diameter of the system can grow infinitely.

- *Arbitrary dynamic churn model.* In this model, there is no restriction on the number of connection and disconnection actions that can occur during any execution of the system. That is, at any time, any subset of processes can join or leave the system and there is no

stabilization constraint on the logical communication graph. This arbitrary dynamic churn model is equivalent to the infinite concurrency model described by Aguilera [5] defined as a model allowing "infinitely many processes, runs have infinitely many processes, and a finite interval time can have infinitely many processes".

1.3 Synchronization

The fundamental problem when devising distributed algorithms is to make all processes cooperate despite the absence of a common notion of time, the presence of failures, adversarial behaviors, churn, locality of the views, or anonymity of the processes.

1.3.1 Synchronization via Causality

The concept of time is fundamental to our way of thinking, [...], however it must be carefully reexamined when considering events in a distributed system' [151]. In a distributed system, the notion of common time does not exist, each entity has its own notion of time, still one may wish entities to do the right thing at the right time. This boils down to allowing entities to synchronize. Synchronization refers to the coordination of entities with respect to time (based on either physical or logical clocks), and to the relative order of events (causality).

For many applications, the relative ordering of events is more important than the actual physical time. *Causality* is a key concept to understand and master the behavior of asynchronous distributed systems. It allows us to determine whether the occurrence of one event is a consequence of the occurrence of another one. Events that are not causally dependent are said to be *concurrent*. All causality relationships that may exist in a distributed execution, that is those that exist between events of the same entity, and those between events of different entities, are completely characterized by the *happened-before* relation defined by Lamport [151]. The happened-before relation, is the smallest transitive relation that satisfies the following conditions.

1. If event e occurred before event f on the same entity, then $e \rightarrow f$
2. If event e is the send event of message and f is the receive event of the same message, then $e \rightarrow f$.

The happened-before relation imposes a partial order on the set of events, and any extension of this partial order to a total order provides an order in which events could have occurred in an execution [127]. Logical clocks [151] provide a way to totally order events. Specifically, a logical clock C is a map from the set of events to the set of natural numbers, such that for any two events e and f, if $e \rightarrow f$ then $C(e) < C(f)$.

Vector clocks [113, 119] on the other hand provide a complete information about the happened-before relation. They have been introduced to allow processes to observe events that are related with the happens-before relation and those that are concurrent. Specifically, a vector clock V is a map from the set of events to vectors of natural numbers, such that for any two events e and f, $e \rightarrow f$ if and only if $V(e) < V(f)$. The timestamp of an event produced by a process is the current value of the vector clock of the corresponding process. In that way, by associating

vector timestamps with events it becomes possible to safely decide whether two events are causally related or not without any synchrony assumption. According to the problem he focuses on, a designer might be interested only in a subset of the events produced by a distributed execution (e.g., only the checkpoint events are meaningful when one is interested in determining consistent global checkpoints). Some applications mainly related to the analysis of distributed computations [101] require to associate with each relevant event only the set of its immediate predecessors, and thus the construction of the lattice of consistent cuts produced by the computation must be constructed. The tracking of immediate predecessors allows an efficient on the fly construction of this lattice. More generally, these applications are interested in the very structure of the causal past. The *Immediate Predecessor Tracking* (IPT) problem consists in determining on the fly and without additional messages the immediate predecessors of relevant events. With Jean-Michel Hélary and Michel Raynal we did a contribution in that sense [32]. We have proposed a family of protocols that provides each relevant event with a timestamp that exactly identifies its immediate predecessors. The family is defined by a general condition that allows application messages to piggyback control information whose size can be smaller than n (the number of processes).

1.3.2 Synchronization via Physical Clocks

Entities may have access to real-time measuring devices by reading the time either given by hardware clocks or across a communication network. In both cases it is necessary to adjust the time read from these sources of time to provide each entity a good approximation of a common time. The construction of synchronized clocks is achieved by *clock synchronization algorithms* whose purpose is to ensure that spatially separated entities have a common knowledge of time. Essentially such an algorithm overcomes clock drift, variations of transmission delays and failures. It guarantees that the maximum deviation between (correct) local clocks is bounded (precision) and that these clocks are within a linear envelope of real-time (accuracy). The clock synchronization problem is an approximate agreement problem [162]. In this problem, entities start with real-valued inputs and eventually decide on real-valued outputs such that any two output values differ by no more than a small positive real-valued tolerance ε. More precisely, a solution to this problem should guarantee the following conditions.

- Agreement. The decision values of any pair of non-faulty[2] entities are within ε of each other
- Validity. Any decision of any non-faulty entity should be in the range of the initial values of the non-faulty entities
- Termination. All non faulty entities eventually [3] decide

Designing clock synchronization algorithms presents a number of difficulties. First, due to (bounded) variations of transmission delays each entity cannot have an instantaneous global view of every remote clock value. Second, even if all clocks could be started at the same real time, they would not remain synchronized because of drifting rates. Cristal clocks run at a rate that can differ from real time by 10^{-5} seconds per second and thus can drift apart by one second per day. In addition, their drift rate can change due to temperature variations or aging. The difference between two hardware

[2]The specification holds for both stopping and Byzantine entity failures

[3]The specification of the approximate agreement problem holds for any synchrony assumptions. Note that the clock synchronization problem makes sense only when message delay are bounded.

clocks can thus change as time passes. Finally, the most important difficulty is to support failures. Therefore hardware clocks have to be synchronized with each other at a regular interval. Clock synchronization has been extensively studied for years, and has given rise to a myriad of algorithms according to the assumptions made on the system. Thorough surveys [184, 193] and [197] exist. With Isabelle Puaut, we have extended these surveys by proposing a taxonomy [43, 44] adapted to all fault-tolerant clock synchronization algorithms— deterministic and probabilistic, internal and external, and resilient from crash to Byzantine failures— with the hope to help application designers in choosing the most appropriate structure of algorithm and the best building blocks suited to his/her hardware architecture, failure model, quality of synchronized clocks and message cost induced. This taxonomy allowed us to compare the behavior of a large range of synchronization algorithms in a common framework. The performance evaluation was achieved through the simulation of a panel of fault-tolerant clock synchronization algorithms. Based on this study, with Carole Delporte-Gallet, Hugues Fauconnier and Josef Widder we have proposed a synchronization algorithm adapted to space applications where exceptionally long service life (several years) are a prerequisite [24,25]. Our solution was based on the idea that permanent failures are too optimistic for certain applications, while fault-tolerant self-stabilization might be too pessimistic, or the provided properties too weak. We therefore explored under which conditions clock properties can be provided permanently in the presence of transient and dynamic Byzantine faults, where processes recover from "bad periods" with an arbitrary state and just start following their algorithm. In particular our algorithm guarantees that in presence of up to f "moving" and concurrent Byzantine failures, correctly behaving processes (that is at least $(n-f)$ processes, with $n \geq (3f+1)$, and n the number of processors in the system) have synchronized logical clocks [26]. Our algorithm is a variation of Srikanth and Toueg's clock synchronization algorithm [205], in which the classic notion of "correct process" is assumed. The challenge of the work was the guarantee that correctly behaving processes are never corrupted by recovering processes, and that clocks of recovering processes get quickly tightly synchronized with those of correctly behaving processes. Actually, we provided a bound for the recovery time (i.e., the period of time after which a recovered process is synchronized with the other processes), which is independent of f, and is roughly equal to the time needed to execute two resynchronizations. The failure turn-over rate, i.e., the maximal allowable frequency at which processes may enter (and leave) faulty periods is also independent of f, and is roughly equal to the time needed to execute three resynchronizations.

1.4 Communication

Communication abstractions permit the broadcasting of a message to a group of processes and offer diverse reliability guarantees for delivering messages to the processes.

In permissioned systems, broadcast primitives guarantee strong properties regarding the delivery of the messages to the recipients and the order in which these messages are delivered [134]. For example, a reliable broadcast primitive guarantees that any message sent to a set of entities is delivered either by all of them or by none of them [134]; an atomic broadcast primitive ensures that the order in which messages are delivered to the recipients is the same for all of them [8, 10, 24, 96, 134], and a causal broadcast primitive guarantees that if any two messages are causally dependent then their delivery order at all recipient is respected [39, 134, 189]. These communication primitives greatly facilitate the task of an application designer since strong properties are guaranteed. For instance,

ensuring the consistency of multiple replicated copies becomes trivial whenever an atomic broadcast service is available.

In permissionless systems, there are mainly two ways for entities to communicate over peer-to-peer networks: through randomized spreading (also called gossip broadcast, or randomized rumor spreading) and through semantic-based broadcast.

1.4.1 Randomized rumor spreading

Randomized rumor spreading or *gossiping* is an important mechanism that allows the dissemination of information in large and complex networks through pairwise interactions. This mechanism initially proposed by [99] for the update of a database replicated at different sites, has then been adopted in many applications. In contrast to reliable communication broadcasts which must provide agreement on the broadcast value with possibly additional ordering guarantees on the delivery of updates from different sources, a randomized rumor spreading primitive provides reliability only with some probability. This probability is determined by the number of rounds for which the message is propagated before it is discarded. The goal of the designer is to analyze the number of rounds until the message is expected to be diffused globally. In a typical randomized rumor spreading protocol, each participating entity is in charge of a part of the dissemination process. When an entity wants to broadcast a message, it selects t entities from the system at random and sends the message to them; upon receiving a message for the first time, each entity repeats this procedure. In order to select a gossip entity at random among all system entities, each entity relies on partial views instead of full membership.

The important question raised by these protocols is the spreading time, that is the time it needs for the rumor to be known by all the entities of the network. Several models have been considered to answer this question. The most studied one is the synchronous push-pull model, also called the synchronous random phone call model. This model assumes that all the entities of the network act in synchrony, which allows the algorithms designed in this model to divide time in synchronized rounds. During each synchronized round, each entity i of the network selects at random one of its neighbor j and either sends to j the rumor if i knows it (push operation) or gets the rumor from j if j knows the rumor (pull operation). In the synchronous model, the spreading time of a rumor is defined as the number of synchronous rounds necessary for all the nodes to know the rumor. In one of the first papers dealing with the push operation only, Frieze [123] proved that when the underlying graph is complete, the ratio of the number of rounds over $\log_2(n)$ converges in probability to $1 + \ln(2)$ when the number n of entities in the graph tends to infinity.

Several authors have recently dropped the synchrony assumption by considering an asynchronous model. In the discrete time case, Acan et al. [3] study the rumor spreading time for any graph topology. They show that both the average and guaranteed spreading time are $\Omega(n \ln(n))$, where n is the number of entities in the network. Angluin et al. [48] analyze the spreading time of a rumor by only considering the push operation (which they call the one-way epidemic operation), and show that with high probability, a rumor injected at some node requires $O(n \ln(n))$ interactions to be spread to all the nodes of the network. This result is interesting, nevertheless the constants arising in the complexity are not determined. In the continuous time case, Ganesh [124] considers the propagation of a rumor when there are n independent unit rate Poisson processes, one associated with each node.

At a time when there is a jump of the Poisson process associated with node i, this node becomes active, and chooses another node j uniformly at random with which to communicate. Ganesh [124] analyzes the mean and the variance of the spreading time of the rumor on general graphs and Panagiotou and Speidel [175] propose a thorough study for spreading a rumor on particular Erdös-Rényi random graphs. In [93], Daley and Kendall propose a different model in which, in addition to spreaders and ignorants, is introduced the notion of stiflers. A stifler learns the rumor but does not propagate it. A stifler results from the interaction between two spreaders, or between a spreader and a stifler. Recently, with Mocquard and Sericola [166, 167] we have considered the rumor spreading time in the asynchronous push-pull model for both the discrete and continuous time cases in the population protocol model, i.e., entities do not have any identifier, and they do not have access to the total number of entities in the system. We gave simple expressions of the expected value and variance of the total number of interactions needed for all the nodes to get the rumor, and an explicit expression of its distribution.

1.4.2 Semantic-based broadcast

This abstraction allows to distribute information to all interested entities within the system, where the latter are determined according to their known interests and the information contained in the message. This communication schema is captured by the *publish/subscribe* paradigm. This paradigm is an effective technique for building distributed applications in which information has to be disseminated from publishers (event producers) to subscribers (event consumers) [6]. The decoupled nature of publish/subscribe interaction makes these architectures particularly suitable for building applications where scalability plays a key role. In publish/subscribe systems, users express their interests in receiving certain types of events by submitting a predicate defined on the event contents. The predicate is called the user subscription. When a new event is generated and published to the system, the publish/subscribe infrastructure is responsible for checking the event against all current subscriptions and delivering it efficiently and reliably to all users whose subscriptions match the event. The challenges that need to be solved are the following ones: *(i)* storing efficiently and reliably all subscriptions associated with the respective subscribers. *(ii)* receiving all relevant events from publishers, and *(iii)* dispatching all published events to the correct subscribers. Combining expressiveness of subscription language and scalability of the infrastructure poses an interesting challenge that has inspired many researchers to explore this topic further.

Typical implementations of publish/subscribe systems, including Siena [76], rely on a network of dedicated servers (usually called brokers) that are controlled by administrators in charge of repairing and maintenance interventions, while for instance Scribe [78] and Bayeux [215] are topic-based systems and rely on a distributed hash table. A single entity is responsible for matching and delivering all notifications related to a specific topic, which is the root of a dynamically-built diffusion tree for events. The main difficulties in designing content-based pub/sub on top of DHTs are *(i)* mapping content-based subscriptions into a single key space and *(ii)* ensuring the persistence of subscriptions despite the dynamicity of the underlying overlay. Methods to map content-based subscriptions to DHT addresses have been described for instance in [62, 207]. The mapping requires subscriptions to be moved from the issuing node to a set of selected “rendezvous" nodes. Mapping may impose some restriction on the constraints applicable in subscriptions with respect to the general language supported by broker-based systems. Typically, string constraints like prefixes and suffixes

cannot be easily mapped to a set of keys. Moreover, subscriptions are usually replicated on several rendezvous nodes, and large subscriptions (i.e., subscriptions that possibly match a large number of events) may have many copies. Replication is also used in order to maintain the persistence of subscriptions [133]. This is required because a subscriber can lose its subscriptions if a rendezvous node fails.

With Datta, Potop Butucaru and Virgilito [21] we have proposed a generic content-based publish/subscribe system, called DPS (*Dynamic Publish/Subscribe*), which is not based on a network of brokers. Subscribers coordinate among themselves on a peer-to-peer basis to construct optimized event diffusion paths without any human intervention. Briefly, we have proposed a subscription-driven semantic overlay in which subscribers self-organize according to similarity relationships among their subscriptions. Similar subscribers are logically connected into the same group. Groups of subscribers self-configure to form tree structures such that only one tree is built per attribute. Virtually all types of attributes and constraints can be directly supported. Differently from some solutions for semantic-driven pub/sub overlay [116], DPS does not assume the complete knowledge of the network to compute the neighbors of an entity. Thus, each subscriber has to keep track of a limited number of its neighbors regardless of the size of the system, and the effect of node failures is confined within a bounded number of neighboring groups. The general design principles of the DPS overlay can be instantiated with different algorithms to i) traverse the tree for propagating a subscription or a publication across the groups and ii) realize the communication inside a group and between groups. Tree traversal and communication approaches can be combined to design DPS implementations that cater the needs of different deployment contexts. In particular, we proposed two different techniques for tree traversals (namely, *root-based* and *generic*) and two different approaches for communication (*leader-based* and *epidemic*).

1.5 Membership Management

Most of the distributed algorithms build their decision value based on the knowledge they have about which processes in the system are currently up or not during the computation. As previously seen, such a knowledge can be locally acquired by any process by invoking its failure detector module. Failure detectors provide suspicions on who is down but whatever the accuracy on the failure status of processes, there is no guarantee that the information provided by any two failure detectors modules are consistent.

From the permissioned computing perspective, and for most of the distributed problems, including consensus, atomic commit, or reliable broadcast, such a level of inconsistency is sufficient to design algorithms where both safety and liveness properties hold. For other problems such as atomic broadcast, instead of relying on failure detectors, the algorithm may rely on a *group membership* service. Such a service is responsible for *(i)* determining the set of processes that are currently up during the computation, and *(ii)* for ensuring that processes agree on the successive values of this set. Such a set of processes is called a *group*. For example, a group may be a set of processes that are cooperating towards a common task (e.g., the primary and backup servers of a database), a set of processes that share a common interest (e.g., clients that subscribe to a particular newsgroup), or the set of all processes in the system that are currently deemed to be operational. Since new processes may want to join an existing group, and others may want to leave or have to be removed

after they fail, the membership of a group changes dynamically. Thus such a service provides consistent information about the long term evolution of the system. It has been widely recognized that the task of giving a precise specification of group membership was either difficult to understand or problematic [20], or too strong to be implemented [81]. As argued by Toueg and Schiper [192], the difficulty came from the mixing of both objectives *(i)* and *(ii)* of group membership. Actually, an important question that has been tackled by Urbán et al. [208] concerns the impact of the group membership service or failure detectors on the performance of long term distributed algorithms such as uniform atomic broadcast algorithms. Their study shows that in absence of crashes and suspicions, both algorithms exhibit the same latency. On the other hand, failure detector-based algorithm should be used to make the algorithm more responsive in presence of crashs and more robust in presence of wrong suspicions, while group membership should be used to get the long term performance and resiliency benefits after a crash.

From a permissionless perpective, it seems unrealistic to deploy applications for which the correction is based on a precise and consistent knowledge of the entities present and absent essentially because of the high dynamism exhibited by these systems, and the fact that entities are interconnected through unreliable channels. Rather, applications rely on the assumption that each participating entity maintains its own local view of the other participants with which it can directly communicate. To keep the system connected despite churn, the renewal of such a view must be periodically achieved. This is achieved through the construction of a *node sampling service*, which is a functionality local to each entity of the system, returns the identifier of a random entity that belongs to the system [70]. This service continuously reads the input stream of the entity. Data streams are made of the entity ids exchanged within the system through gossip-based algorithms or from the ids received during random walks initiated at each entity of the system. The node sampling servicer is a cooperative service in the sense that all the entities of the system contribute to this service by continuously sending and forwarding information about their presence.

Unfortunately, in permissionless systems, the unavoidable presence of malicious nodes seriously impedes the construction of uniform node sampling [141,158,198]. The objective of malicious entities mainly consists in continuously and largely biasing the input data stream out of which samples are obtained (by for instance, infinitely often augmenting it with the ids it manipulates), to prevent (correct) entities from being selected as samples. Consequences of these collective attacks are, among others, the overwhelming load of some specific entities when it is used to provide random locations for data caching or storage, or the eventual partitioning of the system when the node sampling service is used to build entity local views in epidemic-based protocols. With Busnel and Sericola, we have proposed a *robust node sampling service* [18] tolerant to malicious nodes, which guarantees both *Uniformity* and *Freshness* properties. Uniformity states that any entity in the system should have the same probability to appear in the sample of correct entities in the system, while Freshness says that any entity that recurs infinitely often in the stream, should have a non-null probability to appear infinitely often in the sample of any correct entities in the system. Specifically, if $S_i(t)$ denotes the output of the sampling service at any correct entity i[4]. at any discrete time t, then a sampling service tolerant to malicious behaviors should meet the following two properties.

[4]Although malicious entities have also access to a sampling service, we cannot impose any assumptions on how they use it as their behavior can be totally arbitrary.

- Uniformity. For any discrete time $t \geq 0$, for any entity $j \in \mathcal{N}$,

$$\Pr\{S_i(t) = j\} = \frac{1}{n}$$

- Freshness. For any discrete time $t \geq 0$, for any entity $j \in \mathcal{N}$,

$$\{t' > t \mid S_i(t') = j\} \neq \emptyset \text{ with probability 1.}$$

Briefly, solutions that basically consist in storing the identifier of all the entities of the system so that each of these entity identifiers can be randomly selected when needed are impracticable and even infeasible due to the size and the dynamicity of such networks. Rather providing a solution that requires as little space as possible (*e.g.* sublinear in the population size of the system) is definitely desirable. Bortnikov *et al.* [70] have proposed a uniform node sampling algorithm that tolerates malicious nodes by exploiting the properties offered by min-wise permutations. Specifically, the sampling component, which is fed with the stream of node identifiers periodically gossiped by nodes, outputs the node identifier whose image value under the randomly chosen permutation is the smallest value ever encountered. Thus eventually, by the property of min-wise permutation, the sampler converges towards a random sample. However by the very same properties of min-wise permutation functions, once the convergence has been reached, it is stuck to this convergence value independently from any subsequent input values. Thus the sample does not evolve according to the current composition of the system, which makes it static. Actually imposing strict restrictions on the number of messages sent by malicious nodes during a given period of time and providing each correct node with a very large memory (proportional to the size of the system) is a necessary and sufficient condition to output an unbiased and non static stream [15]. As a consequence, lack of adaptivity or full-space algorithms has for a long time seemed to be the only defenses against adversarial behaviors when considering deterministic algorithms.

Together with Busnel, Gambs, and Sericola, we have studied this issue by adopting a probabilistic approach [16, 18]. Briefly we first proposed an omniscient strategy that processes on the fly an unbounded and arbitrarily biased input stream made of node identifiers exchanged within the system, and outputs a stream that preserves Uniformity and Freshness properties. Through a Markov chains analysis we have shown that both properties hold despite any arbitrary bias introduced by the adversary. We have then proposed a randomized approximation algorithm that is capable of outputting an unbiased and non static sample of the population whatever the strategy of the adversary is. This sample may deviate from an exact uniform sample, however the deviation is bounded with any tunable probability. This algorithm is a one-pass algorithm, *i.e.*, each piece of data of the input stream is scanned sequentially on the fly, and only compact synopses or sketches that contain the most important information about data items are locally stored. This algorithm does not require any a priori knowledge neither on the size of the input stream, nor on the number of distinct elements that compose it, nor on the frequency distribution of these elements. We have then evaluated the minimum effort that needs to be exerted by a strong adversary to bias the output stream when two representative attacks are launched, *i.e.*, the *targeted attacks* in which the adversary focuses on biasing the frequency of a single node identifier, and the *flooding attack* which aims at biasing all the node identifiers frequencies. Both evaluations are conducted by modeling them as an urn problem. One of the main results of this analysis is the fact that the effort that needs to be exerted by the adversary to subvert the sampling service can be made arbitrarily large by any correct entity by just increasing the memory space of the sampler.

Note that a robust node sampling service is appropriate for both unstructured and structured logical overlays. This is straightforward for unstructured one since the topology of unstructured overlays conforms with randoms graphs, i.e., relationships among entities are set according to a random process. Regarding structured logical overlays (i.e distributed hash tables (DHTs)), they build their topology according to structured graphs (e.g., hypercube, torus). For most of them, the following principles hold: each entity is assigned a unique random identifier from an m-bit identifiers space (Identifiers are derived by applying some standard cryptographic one-way hash function on peers intrinsic characteristics (e.g., IP address), and the value of m is large enough to make the probability of identifiers collision negligible.) The identifier space is partitioned among all the entities of the overlay, and entities self-organize within the structured graph according to a distance function D based on their identifiers IDs (e.g., two entities are neighbors if their identifiers share some common prefix), plus possibly other criteria such as geographical distance. Thus to be robust against attacks that aim at partitioning the overlay, any entity in the system should have the same probability to appear in the sample of correct nodes among all the entities that satisfy the distance function D.

1.6 Digital Reputation, a Strong Incentive for Cooperation

Reputation systems are becoming attractive for encouraging trust among entities that usually do not know each other. A reputation system collects, distributes, and aggregates feedback about the past behavior of a given entity. The derived reputation score is used to help entities to decide whether a future interaction with that entity is conceivable or not. Without reputation systems, the temptation to act abusively for immediate gain can be stronger than the one of cooperating.

Designing reputation systems in permissionless systems has to face the absence of any large and recognisable but costly organisations capable of assessing the trustworthiness of a service provider. The only viable alternative is to rely on informal social mechanisms for encouraging trustworthy behavior [100]. Proposed mechanisms often adopt the principle that "you trust the people that you know best", just like in the word-of-mouth system, and build transitivity trust structures in which credible peers are selected [211–213]. However such structures rely on the willingness of entities to propagate information. Facing free-riding and more generally under-participation is a well known problem experienced in most open infrastructures [4]. The efficiency and accuracy of a reputation system depends heavily on the amount of feedback it receives from participants. According to a recognised principle in economics, providing rewards is an effective way to improve feedback. However rewarding participation may also increase the incentive for providing false information. Thus there is a trade-off between collecting a sizeable set of information and facing unreliable feedback [72]. An additional problem that needs to be faced with permissionless systems, is that entities attempt to collectively subvert the system. Entities may collude either to discredit the reputation of a provider to lately benefit from it (bad mouthing), or to advertise the quality of service more than its real value to increase their reputation (ballot stuffing). Lot of proposed mechanisms break down if voters collude [97].

With Aina Ravoaja we have studied the robust reputation problem. Essentially this problem aims at motivating entities to send sufficiently honest feedback. This is accomplished by an aggregation technique in which a bounded number of entities randomly selected within the system report di-

rectly observed information to requesting entities. Observations are weighted by a credibility factor locally computed, while incentive for participation is implemented through a fair differential service mechanism relying on entitiy level of participation and entity credibility. We showed that with some modest churn the presence of a high fraction of malicious entities does not prevent a correct entity from accurately evaluating the reputation value of a target entity. However, this is achieved at the expense of a non negligible number of messages mainly due to the way feedback are collected. Indeed, to face malicious entities, feedback is gathered from entities randomly selected within the system which clearly makes the efficiency of this crawling technique highly reliant on the way the graph of witnesses is constructed. Finding the right set of witnesses in an efficient way is a challenging issue since the reputation value depends on the feedback provided by these entities [45]. We have then improved upon this solution by relying on two schemes: first, to increase the likelihood of collecting a large amount of feedback from the right set of witnesses, we need secure gathering techniques. Briefly, these techniques aim at guaranteeing that feedback is collected only from a cluster of entities sharing a similar interest for the target service provider, and that bribes and collusion do not interfere by constraining the gathering scheme. Our platform called Storm [185] has been designed to gather information on the reputation of a peer in $O(\log N)$ steps, with N the number of peers. Complementary to this scheme, secure routing tables maintenance and secure routing techniques have been provided. These techniques ensure first that routing tables cannot be attacked by a non adaptive adversary by keeping the distribution of malicious peers uniform.

In our quest to build robust reputation systems, with Heverson Ribeiro we have addressed the issue of feedback persistency. Reputation systems are clearly concerned with this issue for the simple reason that entities can freely join and leave the system at any time. This continuous churn must not prevent the reputation system to have continuous access to feedback the system has succeeded to gather, evaluate and store at multiple nodes in the system. Beyond churn, the unavoidable presence of malicious nodes may also be an obstacle to guarantee durable access to feedback. Reasons are numerous, and among them one can cite whitewashing, transaction repudiations, bad mouthing or even ballot stuffing. Actually two main techniques exist for handling data redundancy in permissionless systems: *full replication* and *erasure coding*. Replication is based on creating copies of the original data object and placing these copies in distinct places in the system. The main benefits of using this scheme is clearly its ease of implementation and its very low download latency overhead. On the other hand, storage overhead incurred by the storing of full data object replicas and bandwidth needed to recreate new copies upon unpredictable join and leave of nodes tend to overwhelm the benefits of replication. Erasure coding provides redundancy without the overhead of replication. Specifically, an object is divided into k equal size fragments, and recoded into s coded symbols, with $s > k$. The ratio $\frac{k}{s}$ is called *code rate* and it gives the exact amount of redundancy added to the original data. Fundamental property of erasure coding is that the original data is recoverable from any k distinct coded blocks. Reed-Solomon codes, the pioneered fixed-rate erasure codes, are well adapted to bounded-loss channels, while Tornado codes ensure fast coding and decoding operations. However fixed-rate erasure codes are intrinsically not adapted to unbounded-loss channels because the generated coded blocks are interdependent. Rateless codes, also called Fountain codes, overcome this feature. As a class of erasure codes, they provide natural resilience to losses, and therefore are fully adapted to dynamic systems. By being rateless, they give rise to the generation of random, and potentially unlimited number of uniquely coded symbols. This makes content reconciliation useless and one may recover an initial object by collecting coded blocks generated by different sources. The three main rateless codes are the pioneering LT codes introduced by Luby [160], the online codes by

Maymounkov et al. [164] and Raptor codes by Shokrollahi [196]. The two latest ones independently discovered the idea of adding a pre-coding phase to obtain linear codes and stronger recoverability guarantees. Empirical studies have shown that data redundancy management (full replication vs. erasure coding) strongly depends on both nodes availability and rate at which these changes take place. It has been shown [190] that erasure coding benefits vanish due to their implementation complexity and the increase in terms of download latency. With Heverson, we have shown that by judiciously managing full replication and coding one can keep the best of both techniques in permissionless systems [186–188].

While aggregated ratings are necessary to derive reputation scores, identifiers and ratings are personal data, whose collect and use may fall under legislation [111]. Unfortunately, solely relying on pseudonyms to interact is not sufficient to guarantee user privacy [206]. This has given rise to the proposition of a series of reputation mechanisms which address either the non-exposure of the history of raters (e.g., [68]), the non-disclosure of individual feedback (e.g., [135, 177]), the secrecy of ratings and the k-anonymity of ratees (e.g., [90]), or the anonymity and unlinkability of both raters and ratees (e.g., [47, 68]). Regrettably, the search for privacy has led to restrictions in the computation of the reputation score: clients cannot issue negative ratings anymore [47, 68]. This restriction comes from the management of ratings. In existing privacy-preserving mechanisms, the ratees have the opportunity to skip some of the received ratings to increase their privacy. However, this is not conceivable in a non-monotonic reputation mechanism: ratees could skip negative ratings to increase their reputation scores. This is a problem because a ratee that received a thousand positive ratings and no negative ratings is indistinguishable from a ratee that received a thousand positive ratings and a thousand negative ratings. This clearly does not encourage ratees to behave correctly. Furthermore, Baumeister *et al.* explain that "bad feedback has stronger effects than good feedback" on our opinions [63]. Negative ratings are thus essential to reputation mechanisms. So far, preserving the privacy of both raters and ratees and handling both positive and negative ratings has been recognized as a complex challenge. Quoting Bethencourt, "Most importantly, how can we support non-monotonic reputation systems, which can express and enforce bad reputation as well as good? Answering this question will require innovative definitions as well as cryptographic constructions" [68].

The design of a reputation mechanism preserving the privacy of both raters and ratees, and allowing for both positive and negative ratings is however highly desirable in many applications where both quality of service and privacy are very important. This is particularly true in all audit processes, and web-based community applications. As an example, let us focus on an application that concerns all of us. Consider the scientific peer review process. This process is at the core of many scientific committees, including those of scientific conferences, journals, and grant applications. Its goal is to assess the quality of research through experts, the peer reviewers, that evaluate manuscripts and proposals based on their scientific quality, significance, and originality. Peer reviewing is an activity that requires to spend considerable effort and time for doing legitimate, rigorous and ethical reviews. Reviewers are chosen by committee members among all their colleagues and peers. In order to remove any bias, several reviewers are assigned to each manuscript, and the reviews are double-blind. The increasing number of solicitations makes the reviewing task even more challenging, and by force of circumstances may lower the quality of reviews. Imagine now a privacy-preserving reputation mechanism whose goal would be to assess reviews according to their helpfulness and fairness. Authors would anonymously rate each received review according to both criteria. Journal editors or

committee chairs would collect these ratings to update the reputation score of the concerned reviewers. Those reputations would be maintained in a very large shared anonymous repository organized according to the thematic of the anonymized reviewers. Editors and chairs would then have the opportunity to choose anonymous reviewers from this repository according to their reputation to build their reviewing committee. The rationale of such a repository would be three-fold. Authors would have an incentive to honestly and carefully rate each received review as their goal would be to contribute to the creation of a pool of helpful and fair reviewers. Committee chairs (*i.e.* editorial boards, program committee chairs, grant application boards) would take advantage of using such a repository as they could decrease the load imposed to each selected reviewer by soliciting a very large number of them. Consequently, each reviewer could devote more time to their few reviews, and would thus provide highly helpful reviews, increasing accordingly the quality of published articles or granted applications. Finally, it would be at reviewers' advantage to provide helpful reviews to have a high reputation. Indeed, being solicited through this reputation mechanism would be an unanimous acknowledgement of their expertise. As a consequence, such an anonymous repository would increase the quality of accepted manuscripts and proposals in a fully privacy-preserving way. With Paul Lajoie-Mazenc, Gilles Guette, Thomas Sirvent, Valérie Viet Triem Tong, we have addressed this problem by proposing the design and security evaluation of a non-monotonic distributed reputation mechanism preserving the privacy of both parties [30, 31, 148, 149]. Briefly, our reputation mechanism aims at offering three main guarantees to both clients and service providers. First and foremost, that the privacy of both parties is preserved; second, that reports cannot be prevented from being issued and finally, that every data needed for the computation of reputation scores is available and cannot be falsified. These guarantees have been partitioned in privacy properties, which ensure that clients do not know the service providers with which they interact up to the point they rate them (inclusive), and that clients are always anonymous among all clients, that is, two clients are indistinguishable, i.e.,

- *Privacy of service providers* When a client rates an honest service provider, this service provider is anonymous among all honest service providers with an equivalent reputation.
- *Privacy of clients* When a provider conducts a transaction with an honest client, this client is anonymous among all honest clients. Furthermore, the interactions of honest clients with different providers are unlinkable.

Safety properties guarantee that computation of the reputation scores cannot be biased by ballot-stuffing attacks, and that reputation scores are unforgeable.

- *Linkability of reports* Two valid reports emitted by the same client on the same service provider are publicly linkable.
- *Unforgeability of reputation scores* A provider cannot forge a valid reputation score different from the one computed from all the reports assigned to this provider.
- *Unforgeability of reports* If a report involving a client and a service provider is valid and either the client or the provider is honest, then this report was issued at the end of an interaction between both users.

Finally, liveness properties guarantee that clients cannot prevent providers from obtaining a proof of transaction, and that providers cannot prevent clients from posting ratings, and that reports are unforgeable, i.e.,

- *Undeniability of ratings* At the end of a transaction between a client and a provider, the client can issue a valid rating, which will be taken into account in the reputation score of the provider.
- *Undeniability of proofs of transaction* At the end of a transaction between a client and a provider, the provider can obtain a valid proof of transaction.

To cope with the fact that service providers must not manage themselves their reputation score to guarantee the absence of fraudulent manipulation, a distributed trusted authority, populated by *accredited signers* is in charge of updating and certifying reputation scores. The main features of this distributed authority are the following ones. Firstly, this authority must involve fairly trusted entities or enough entities to guarantee that the malicious behavior of some of them never compromises the computation of reputation scores. Secondly, they must ensure that providers remain undistinguishable from each others, which cannot be achieved (in a simple way) when providers are managed by different authorities. Thus, a unique and fairly trusted or reasonably large set of accredited signers manages all reputation scores.

Moreover, by the undeniability properties, a client or a provider must be able to cast a report, even if the other party does not complete the interaction. To guarantee the privacy of this party, no identifying data can be sent before the transaction. However this is obviously necessary to cast a report. To solve this issue, a distributed trusted authority, populated by *share carriers*, is in charge of guaranteeing that reports can be built. This distributed authority must collect information before the transaction, and potentially help one of the two parties afterwards, consequently it must thus be online. Both distributed authorities could be gathered in a single one. The drawback of this approach is that this distributed trusted authority should be simultaneously online, unique, and fairly trusted or reasonably large. The unicity and the participation in each interaction would induce an excessive load on each entity of this distributed authority. We thus suggest distinct authorities, for efficiency reasons. Accredited signers are then a unique set of fairly trusted or numerous entities, periodically updating the reputation scores of all providers. Share carriers are on the other hand chosen dynamically during each interaction among all the service providers. Since they are responsible for the availability of a single report, share carriers do not need to be as trustworthy as the accredited signers.

To deal with privacy of both clients and providers, share carriers use *Verifiable Secret Sharing* [117]. This basically consists in disseminating the shares of a secret to the share carriers, so that they cannot individually recover the secret, but allow the collaborative reconstruction of this secret.

1.7 Consensus

The Consensus abstraction is at the core of most of the distributed algorithms. The consensus problem is a coordination problem, where n parties attempt to reach agreement on a value from some fixed domain V, despite the malicious behavior of up to t of them. More specifically, every party starts the consensus protocol with an initial value $v \in V$, and every execution of the protocol must satisfy (except possibly for some negligible probability) the following conditions:

- Termination: All honest parties decide on a value.

- Agreement: If two honest parties decide on v and w, respectively, then $v = w$.
- Validity: If all honest parties have the same initial value v, then all honest parties decide on v.

The domain V can be arbitrary, but frequently the case $V = \{0, 1\}$ is considered given the efficient transformation of binary agreement protocols to the multi-valued case. There exist various measures of quality of a consensus protocol: its resiliency, expressed as the fraction t/n of misbehaving parties a protocol can tolerate; its running time—worst number of rounds/steps by which honest parties terminate; and its communication complexity—worst total number of bits/ messages communicated during a protocol execution. The above definition captures the classical definition of the consensus problem. A related and extensively studied version of the problem is *state-machine replication* or "Nakamoto" consensus that I will treat in Section 1.8.

The consensus problem has been extensively studied from both a theoretical point of view and a practical one for more than thirty years. It has given rise to many positive and negative results depending on the assumptions made on the system and the specification of the problem. In terms of specifications it is strongly related to other distributed problems such as atomic broadcast [134] also called state machine replication [152] and group membership [20, 87]. Those relationships are important since many results that apply to consensus implementability also apply to those distributed algorithms [92]. From a theoretical point of view, the seminal works of Pease, Shostak, and Lamport in 1980 on interactive consistency [178], followed by the Byzantine generals problem [156] has played a key role in the emergence of a whole new field of research, centered around formalizing and characterizing various consensus problems. They showed that it is impossible to achieve consensus in the presence of one third of Byzantine processes in the unauthenticated Byzantine model (even when assuming synchrony). This impossibility is not due to the computing power of the individual processes, but rather to the difficulty of coordination between the different processes that compose the distributed system. Fisher, Lynch and Paterson impossibility result [120] (referred to as the "FLP impossibility result") is the milestone that inspired research on the minimal synchrony assumptions necessary to be able to reach consensus deterministically. The FLP result shows that one cannot solve consensus in a completely asynchronous environment with reliable links even if a single process may crash [134]. Consequently, to circumvent the FLP impossibility result, the research community has focused on strengthening the system model by introducing additional assumptions on the system synchrony (e.g, [80, 102, 106, 147, 163]), by relying on failure detectors (e.g., [9, 10, 82, 122]), or by relaxing the problem definition so that that the solution terminates with high probability, through common coin (e.g, [75, 122, 165, 183]), through local randomness (e.g, [64, 204]), or network randomness (e.g, [71]). Cachin et al. [74] provide a relatively recent overview of the many variants of consensus that are considered in the distributed systems literature.

Most of these consensus algorithms have been designed with the objective of reducing the number of communication rounds the algorithm performs, or the number of message exchange steps in case of a non-synchronous system (enriched with failure detectors for asynchronous ones) [142]. The reason is that this time complexity metric is directly related to the decision time.

From the permissionless computing perpective the message complexity, which is defined as the total number of all protocol messages that correct entities generate, highly matters since it is simply not practical to have an algorithm in which the number of messages sent during each round/step

scales linearly with the number of (correct) entities [5]. A lower bound quadratic in the number of participants on the number of messages was shown by Dolev and Reischuk to solve consensus [103]. Consequently, a general idea to deal with the permissionless setting is to execute (traditional) consensus algorithms (with a polynomial message complexity for Byzantine resilient ones (e.g, [75, 80, 107, 163, 165]) on a polylogarithmic (or even sublinear) number of the entities of the system (often called a *committee, cluster, or shard*)). Then, assuming a majority of correct committee members, every entity outside the committee can safely accept the value decided by the committee once a majority of equal values where received. Indeed, if the members of the committee are randomly chosen and independently of the ones initially chosen by a non-adaptive adversary then by applying Chernoff bounds, the number of entities corrupted in the committee approximates the overall fraction of corrupted entities in the full system, which justifies such solutions. Different solutions have been proposed to organise entities of the system in such a way that despite high churn or collusion of entities, consensus algorithms can still be executed without violating the safety of consensus [11, 57, 77, 159, 185, 199].

In particular with Aina Ravoaja, Francesco Brasileiro and Romaric Ludinard we have proposed to structure the entities of the system in a logical overlay which, through its join and leave operations, can handle targeted attacks. These attacks aim at exhausting key resources of targeted hosts (e.g., bandwidth, CPU processing, TCP connection resources) to diminish their capacity to provide or receive services, at preventing data indexed at targeted entities from being discovered and retrieved by poisoning their routing tables, or simply at rerouting or dropping messages addressed to targeted entities. Briefly, PeerCube [11] is a Distributed Hash Table (DHT) that conforms to an hypercube. In PeerCube, each vertex of the hypercube is dynamically formed by gathering entities that are logically close to each other according to a distance function applied on the identifier space. By having entities gathered into quorums (also called shards or clusters), one can introduce the unpredictability required to deal with attacks through randomized algorithms, which is definitively not possible in traditional DHTs. Indeed, relying on single entities to ensure the system connectivity and the correct retrieval of data is not sufficient, since, by holding a logarithmic number of entity ids, the adversary can in a linear number of trials disconnect any entity from the overlay. Shards are built so that the respective common prefix of their members is never a prefix of one-another. This guarantees that each shard has a unique common prefix, that in turn serves as a shard's *label*. The shard's label characterizes the position of the shard in the overall hypercubic topology, as in a regular DHT. Shard size is upper and lower bounded. Whenever the size of shard $\mathcal{S}$ exceeds a given value S_{max}, $\mathcal{S}$ splits into two shards such that the label of each of these two new shards is prefixed by $\mathcal{S}$ label, and whenever the size of $\mathcal{S}$ falls under a given value S_{min}, $\mathcal{S}$ merges with another shard to give rise to a new shard whose label is a prefix of $\mathcal{S}$ label. Each shard self-organizes into two sets, the core set and the spare set. The core set is a fixed-size random subset of the whole shard. It is responsible for running traditional Byzantine agreement protocols (e.g., [75, 80, 107, 163, 165]) in order to guarantee that each shard behaves as a single and correct entity (by for example forwarding all the join and lookup requests to their destination) despite malicious participants. The delivery of network messages is at the discretion of the adversary. To prevent messages from being misrouted or dropped, routing failure tests and redundant routing are used as advocated by Castro et al. [77] and Sit and Morris [199]. Note that such redundant routing have also been successfully implemented in

[5] In the Byzantine setting, one cannot prevent the adversary from sending out of the blue messages to correct entities, thus restricting the communication complexity to messages generated by honest parties seems to be the best one can say about a protocol in a Byzantine environment.

other structured-based overlays (e.g., [118, 139]).

Our construction in itself makes no synchrony assumption except for what is required for the Byzantine resilient algorithms. Members of the spare set merely keep track of shard state. Joining the core set only happens when some existing core member leaves, in which case the new member of the core set is randomly elected among the spare set. By doing this, entities joining the system weakly impact the topology of the hypercube [11]. Still this is not sufficient to defend the system against an adaptive adversary which can corrupt committee members after having observed who they are.

Awerbuch and Scheideler have shown that large-scale systems can survive these attacks only if malicious entities are not able to isolate honest entities within the system [57]. This is achieved by *i)* preserving randomness of entities identifiers, and *ii)* limiting the period of time where entities can stay at the same position in the overlay. *Induced churn* has been shown to be a fundamental ingredient to preserve randomness. Induced churn refers to the general idea of forcing entities to move within the system. Several strategies based on this principle have been proposed. Some strategies, based on *global induced churn*, force each entity to periodically leave and re-join the system. This may be enforced through entity limited lifetime in the system. If not properly handled, these solutions keep the system in an unnecessary hyper-activity, which increases accordingly the impact of churn. Strategies based on *local induced churn* are prefered. However either they were proven incorrect or they involve a too high level of complexity to be practically acceptable [58]. Awerbuch et al [58] proposed the *Cuckoo&flip* strategy. This strategy consists in introducing local induced churn (i.e., forcing a subset of entities to leave the overlay) upon each join and leave operation. This strategy prevents malicious entities from predicting what is going to be the state of the overlay after a given sequence of join and leave operations. Subsequently to this work, experiments have been conducted to check the feasibility of global induced churn. These experiments assume that the overlay is populated by no more than 25% of compromised entities [91], or that the topology of the overlay is static [12, 13].

To implement both limited sojourn time of the entities at the same place in the overlay and unpredictable identifier assignment in a cluster-based overlay, we proposed to limit the lifetime of entity identifiers and randomize their computation [14]. Specifically, entity identifiers are initially generated based on certificates acquired at trustworthy Certification Authorities (CAs). Initial identifiers (denoted id^0) are generated as the result of applying a hash function $\mathcal{H}$ to some of the fields of a X.509 certificate. To enforce all entities, including malicious ones, to leave and rejoin the system from time to time, the creation date t_0 is added to the fields that appear in the entity certificate that is hashed to generate the entity identifier (note that by the properties of hash functions, this guarantees that entity identifiers are unpredictable). The lifetime of entity identifier is limited through an incarnation number. The current incarnation k of any entity is given by the following expression $k = \lceil (t - t_0)/L \rceil$, where t_0 is the initial creation time of the entity's certificate, t is the current time, and L is the length of the lifetime of entities incarnation. Thus, the kth incarnation of an entity p expires when p local clock reads $t_0 + kL$. At this time p must rejoin the system using its $(k+1)^{th}$ incarnation. The t_0 parameter is one of the fields in the entity's certificate and, since certificates are signed by the CA, it cannot be unnoticeably modified by a malicious entity. Moreover, a certificate commonly contains the public key of the certified entity. This way, messages exchanged by the entities can be signed using the sender private key, preventing malicious entities from unnoticeably altering messages originated from others in the system. Messages must contain the certificate of

their issuer, so as to allow recipients to validate them. Therefore, at any time, any entity can check the validity of the identifier of any other entity q in the system, by simply calculating the current incarnation k of entity q and generating the corresponding identifier.

To protect the system against the presence of collusive entities, we have proposed to take advantage of the separation of entity role at the shard level to design robust operations. Briefly, the `join` operation has been designed so that brute force denial of service attacks are discouraged, and the `Leave` operation impedes the adversary from predicting what is going to be the composition of the core set after a given sequence of join and leave events triggered by its malicious entities. Evaluation of our architecture has been conducted in presence of adversarial strategies and has shown two interesting results. By choosing an adequate value for the length of the lifetime of entity incarnation, it is possible to noticeably reduce the propagation of attacks in the whole system, and to remarkably decrease the overhead of the induced churn at cluster level, which demonstrates that there is no need to keep the system in an hyper-activity to be resilient against targeted attacks. Pushing entities smoothly but to unpredictable regions of the system is sufficient [14,19,36,37].

1.8 Nakamoto Consensus

As previously seen, in the consensus problem, processes have initial values which they propose, and have to decide on a single value (this is a " one-shot problem"). Now, for fault tolerance reasons, a service can be deployed over (geographically) distinct servers (called *replicas*), such that clients of the service issue a series of commands to all replicas, which execute them in a coordinated way and produce a reply. This schema of replication is called *state machine* or *active replication* [150].

State machine replication (RSM) is today the foundation of many cloud-based highly-available products: it allows some service to be deployed such to guarantee its correct functioning despite possible faults. In RSM, clients issue operation requests to a set of distributed processes implementing the replicated service, that, in turn, run a protocol to decide the order of execution of incoming operations and provide clients with outputs. Faults can be accidental (e.g. a computer crashing due to a loss of power) or have a malicious intent (e.g. a compromised server). Whichever is the chosen fault model, RSM has proven to be a reliable and effective solution for the deployment of dependable services.

The properties that must be guaranteed when implementing state machine replication (SMR) have been formalized by Schneider [194] as follows:

- Initial state. All correct replica start in the same state
- Determinism. All correct replicas receiving the same input on the same state produce the same output and resulting state.
- Coordination. All correct replicas process the same sequence of commands

RSM is usually built on top of a Consensus primitive that is used by processes to agree on the order of execution of requests concurrently issued by clients. Thus, from a computational point of view, SMR, consensus and atomic broadcast are equivalent [155] ; SMR can be viewed as a sequence of consensus instances so that each value output from a replica corresponds to a command in the SMR log. From a complexity point of view, Antoniadis et al [51] have shown that in a specific

model, completing an SMR command can be more expensive than solving a consensus instance whatever the underlying timing assumptions, which in practice occurs when replicas recover from a failure [67].

Because Consensus does not guarantee progress in asynchronous environments, this led the research community to study and develop alternative solutions based on the relaxation of some of the constraints, to allow agreement to be reached in partially synchronous systems with faulty processes by trading off consistency with availability. Hence most practical fault-tolerant replicated state machines protocols for permissioned systems are built around the Paxos agreement protocol [154] or Raft [174] to tolerate crash failures, or for example on PBFT [79] and Zyzzyva [146] algorithms to be resilient to Byzantine behaviors. They assume a partial synchrony model, rely on a combination of primary and backup replicas to order and execute requests: the leader (sometimes called sequencer or primary) assigns the order for client requests while the backup replicas verify and accept the order defined by the leader, and also monitor the leader to detect its possible crash [154,174], or its Byzantine behavior (e.g., PBFT [79], Zyzzyva [146]). A nice comprehensive survey is presented by Bessani and its co-authors in [67].

From the permissionless computing perspective, the SMR specification must certainly be revisited. Building a replicated log of commands issued by distrustful clients over an unknown and unbounded number of unstable and mutually distrustful participating nodes interconnected by an unpredictable peer-to-peer network may appear as an unrealistic goal!

Yet, in 2008, a eight-pages long white paper published under the pseudonym of Satoshi Nakamoto [173] describes in simple words an astonishing algorithm, called Bitcoin, which allows to reach agreement at any time on a shared chain of blocks, in the presence of a remarkably high proportion of malicious behaviour. This algorithm is the very first one capable to build a robust and secure fully decentralized money system (i.e., that does not require any trusted third parties), without imposing any knowledge on the identity of the participants, and with very weak synchrony assumptions. The fundamental idea of Bitcoin is to integrate within its design the simple fact that nodes behave rationally. Since 2008, this paper has revealed and stimulated a new and growing demand for permissionless distributed algorithms, which is promising since this demand comes from both the academy and the industry.

Nakamoto's solution relies on abstractions that have been studied, for some of them, for more than thirty years, however the scientific breakthrough of his solution is the way it orchestrates them. This solution combines *(i)* a peer-to-peer network, *(ii)* distributed algorithms, *(iii)* cryptographic functions, and least but not last *(iv)* robust incentives to push every participant to participate to the construction of an immutable and continuously growing list of records that mimics the functioning of a unique traditional ledger. In Bitcoin parlance, the *blockchain*.

Prior to formalizing the problem solved by Nakamoto, I would like to briefly describe the proposed solution, which will make the specification of the problem straightforward.

The adversary assumption made by Nakamoto bounds the proportion of the computational power owned by malicious parties. This model called by Abraham and Malki [2] the *Computational Threshold Adversary* is an alternative to the common *Threshold Adversary Model* which bounds the total number of parties the adversary controls relative to the total population of the system. In

Nakamoto's solution, the adversary owns less than 50% of the computational power of the system. As will be clarified below, computation power of the participating modes is used as a mean for minting crypto-currency.

At its core, the Satoshi Nakamoto Consensus protocol combines four elegant and powerful ideas: the UTXO model, a randomized election implemented through a proof-of-work (PoW), a Fork Choice Rule (FCR) strategy, and a short latency broadcast primitive.

- The Bitcoin Unspent Transaction Output (UTXO) model can be roughly seen as a user's account credited by some stake. An UTXO is uniquely characterized by a public key pk_i and its associated amount of stake s_i. Each public key is related to the digital signature schema Σ with the uniqueness property, which allows stakeholders to use the public keys (or a hash thereof) of their UTXOs as a reference to them, as demonstrated in the "Public Keys as Identities principle" of Chaum [86]. At any time, a user can own multiple UTXOs. UTXOs can be debited only once, and once debited, an UTXO does not exist anymore. This model has been introduced by Nakamoto [173], and today is used in the vast majority of the crypto-currency systems. It is a very powerful idea to defend the system against double spending attacks, attacks in which a user spends the same coin in more than one transaction.

- From a functional point of view, a proof-of-work is a cryptographic primitive that enables a verifier to be convinced that a certain amount of work has been invested with respect to a given context, e.g., a plaintext message or a nonce that the verifier has provided [125]. The Satoshi Nakamoto Consensus protocol implements a PoW as follows: all nodes wishing to create the next block of the blockchain engage into a race in order to be the first one (i.e., the leader) to create and to disseminate a valid block containing a solution to a cryptographic puzzle. Once nodes hear of a valid block, they engage in a new race to create the next block of their local blockchain. The strategy to incentive nodes to use their computational power to create blocks is to offer to the winner of the competition a financial reward, which is composed of a fixed part— this is where the money is minted— and a variable one coming from the transaction fees of the block. The strategy to incentive nodes to build valid blocks is to delay the ownership of this financial reward, i.e., the winner will be able to spend her reward only if her block is committed by one hundred subsequent winners.

- The short latency broadcast primitive (gossip primitive) guarantees that the ratio between the message transmission delay and the block time interval remains low enough whatever the system activity, guaranteeing accordingly an easy management of conflicting blocks, if any.

- The Fork Choice Rule (FCR) function provides a decidable strict total order on all possible chains of blocks. Indeed, as such and while the PoW is purposely made difficult, [6] by itself the PoW does not allow nodes to agree on the same sequence of blocks. They can receive valid but conflicting blocks (blocks which point back to the same block) and append them in their local ledgers. Thus, assuming that initially all nodes share the same initial block (the so-called Genesis Block), at any further state of the network, any two nodes' blockchain can be in a fork relation, when neither of them is a prefix of the other. The Fork Choice Rule (FCR) function consists in locally selecting the chain from the tree of chains that has required the greatest amount of computation. It can be shown that by the random nature of the PoW,

[6] The computational hardness or probabilistic rarity of minting valid blocks is what controls the frequency at which blocks are created.

forks cannot last forever [191], i.e., eventually one blockchain will be extended further than the other ones. This is this blockchain that miners will continue to extend.

The problem solved by Nakamoto, the *Nakamoto Consensus* or the *Ledger consensus* [125] can be formalized as follows.

- Uniqueness. There is a unique chain of blocks (i.e., the blockchain) extracted from the tree of chains
- Liveness. The blockchain is constantly growing
- Safety. The deeper a block is buried in the blockchain the harder it is to abort it
- Fairness. The proportion of blocks of non-faulty parties in the blockchain is proportional to their computation power

Links between the distributed computing theory and distributed ledger has been pioneered by Garay et *al* [126], in which they characterized Bitcoin blockchain via its quality and its common prefix properties. Specifically, they have shown that, by keeping the creation rate of blocks very low with respect to their propagation time in the network *(i)* if the adversary controls no more than 1/3 of the network hashing power then it provably controls less than a majority of the blocks in the blockchain and, *(ii)* if the adversary controls no more than 1/2 of the network hashing power then the blockchains maintained by any two honest nodes possess a large common prefix (up to the last k appended blocks), and the probability that they are not mutual prefix of each other decays exponentially in k. Then, a series of analysis have been achieved including [2,33,73,126,138,176] has so far been the distributed ledger agreement aspects. In particular the study conducted in Abraham and Malki [2] that have identified the links between state machine replication and Nakamoto consensus; Pass et al. [176] that have identified additional security properties guaranteed by Bitcoin blockchain; Anceaume et al [33] that have studied solutions that rely exclusively on miners to take in charge the full process of validation and confirmation as for example in [94,112,144]). They have shown that beyond the complexity introduced by these approaches, all important decisions of Bitcoin are put solely under the responsibility of (a quorum of) miners, and the membership of the quorum is decided by the quorum members, which magnifies the power of malicious miners. A series of comprehensive surveys appeared, in particular the one of Bonneau et al. [69] in which Bitcoin blockchain is deconstructed.

Chapter 2

Perspectives

2.1 Consistency Properties of Distributed Ledgers

Distributed ledgers, beyond their incontestable qualities such as decentralisation, simple design and relatively easy use, are neither riskless nor free of limitation. An increasing number of areas promote the use of distributed ledgers for the development of their applications, and undeniably, the properties enjoyed by these technologies should be studied to fit such applications requirements, together with their relationships with blockchain-based applications. Obviously, when distributed ledgers will become the core of strategic applications, the fundamental questions that will raise concern the (minimal) properties that should enjoy a blockchain for a particular type of applications, and the interactions a particular blockchain should have with others blockchains. Such challenges can be mitigated by laying down the consistency properties of distributed ledgers.

In a preliminary work, together with Romaric Ludinard, Maria Potop Butucaru and Frédéric Tronel we have started to investigate the connections between distributed ledgers and read-write registers. This connection is important because it will help us for making possible to argue about the correctness of the ledger itself but also for the applications that will use it. We thus do need to deeply understand and formulate the properties of the state of distributed ledgers.

Briefly, a distributed read-write register is a shared variable accessed by a set of processes through two operations, namely $\mathsf{write}()$ and $\mathsf{read}()$. Informally, the $\mathsf{write}()$ operation updates the value stored in the shared variable while the $\mathsf{read}()$ obtains the value contained in the shared variable. Every operation issued on a register is, generally, not instantaneous and can be characterized by two events occurring at its boundaries: an *invocation* event and a *reply* event. Both events occur at two different instants with respect to the fictional global time: the invocation event of an operation op (i.e., $op = \mathsf{write}()$ or $op = \mathsf{read}()$) occurs at the invocation time denoted by $t_B(op)$ and the reply event of op occurs at the reply time denoted by $t_E(op)$. Given two operations op and op' on a register, we say that op *precedes* op' ($op \prec op'$) if and only if $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* (noted $op||op'$). An operation op is *terminated* if both the invocation event and the reply event occurred (*i.e.*, the process executing the operation does not crash between the invocation time and the reply time). A terminated operation can either be successful and thus returns *true* or can return *abort* when, for example, some operational conditions are not met. In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous.

Depending on the semantics of the operations, three types of registers have been defined by Lamport [153], namely *safe*, *regular* and *atomic* registers. The *safe* register ensures that a read which does not overlap with a write returns the last completed write. The result of a read overlapping a write can be any value from the register domain. The *regular* register verifies the safe semantics when

reads are not concurrent with writes. For reads concurrent with writes the read will return either the last written value or the value of the concurrent write. A *safe* distributed register is defined by the following properties:

- *Liveness*: Any invocation of write() or read() eventually terminates.
- *Safety*: A read() operation returns the last value written before its invocation (*i.e.* the value written by the latest write() preceding this read() operation), or any value of the register domain in case the read() operation is concurrent to a write() operation.

A *regular* distributed register satisfies the liveness property of the safe register and the safety property defined as follows:

- *Safety*: A read() operation returns the last value written before its invocation (*i.e.* the value written by the latest write() preceding this read() operation), or a value written by a write() operation concurrent with it.

An *atomic* register is a regular register that satisfies the no new/old inversion property defined as follows:

- *no new/old inversion:* For any two read operations, the set of writes that do not strictly follow either of them must be perceived by both reads as occurring in the same order.

Interestingly enough, neither safety, regularity nor atomicity of distributed shared registers capture the behaviour of distributed ledgers. Classically, values written in a register are potentially independent, and during the execution, the size of the register remains the same. In contrast, a new block cannot be written in the blockchain if it does not depend on the previous one, and successive writings in the blockchain increase its size.

The behavior of blockchain shares some similarities with stabilizing registers, line of research pioneered by [140]. Looking at the stabilizing register, it implements some type of eventual consistency, in the sense that, there exists a prefix of the system execution for which there are no guarantees on the value of the shared register: register semantics hold only from a certain time in the execution. In contrast, the prefix of the blockchain eventually converges at every entity, while no guarantees hold for the last created blocks. Girault and his colleagues [129] present an implementation of the Monotonic Prefix Consistency (MPC) criterion and showed that no criterion stronger than MPC can be implemented in a partition-prone message-passing system. On the other hand, the proposed formalization does not propose weaker consistency semantics more suitable for proof-of-work blockchains as BitCoin. In their objective to contribute in the provision of a proper formulation of distributed ledgers in terms of shared objects, Anta and his colleagues [50] have proposed a formalization of distributed ledgers modeled as an ordered list of records. The authors have proposed three consistency criteria: eventual consistency, sequential consistency and linearizability. Interestingly, they showed that a distributed ledger that provides eventual consistency can be used to solve the consensus problem. This work has been continued in [49] by defining Multi-Distributed Ledger Objects (MDLO), which is the result of aggregating multiple Distributed Ledger Objects satisfying the definition proposed in [50].

It is already known that some blockchain implementations solve eventual consistency of an append-only queue using Consensus [50]. However, Bitcoin and Ethereum [210] that technically do not solve Consensus need to be characterized from the point of view of the consistency guarantees that they are

able to offer. The question is about the consistency criterion of blockchains as Bitcoin and Ethereum that technically do not solve Consensus, and their relation with Consensus in general.

We are currently addressing this open question by proposing to connect distributed registers theory to distributed ledgers. We have proposed a new register called the *Distributed Ledger Register* (DLR). The Distributed Ledger Register (DLR) has a tree structure, whose root is the genesis block, and where each branch is a sequence of blocks. The *value* of DLR is the best chain of the tree. The best chain is a chain in the tree that satisfies some predicate: the longest sequence of blocks, starting from the root (as in the case of Bitcoin) or the heaviest chain the tree (as in the case of Ethereum). The value of the DLR is called a *blockchain*. The DLR is equipped with write and read operations. The write operation allows any process to append to the value of DLR a block b. The read() operation allows any miner to retrieve the value of DLR. As preliminary results, we have shown that that the state of popular blockchain ledgers satisfy regularity properties [35]. Specifically,

- *Liveness*: Any invocation of write() or read() eventually terminates.
- *Safety*: A read() operation returns a chain $\mathcal{B}$ such that $\mathcal{B}$ has a prefix $\mathcal{B}_1$ whose last($\mathcal{B}_1$) is the last value written before its invocation (*i.e.* the value written by the latest write() preceding this read() operation), or a value written by a write() operation concurrent with it.

By adopting an approach similar to the one used to extend safe registers to atomic ones [56], we intend to provide a construction that provides the distributed ledger register with atomic semantics. Indeed, regular registers provide weak guarantees that are not easy for the application programmer to use for building correct applications on top of those registers. More significantly they do not compose. Atomicity or linearizability is a state property that is composable [136], and thus would allow the combination of multiple distributed ledgers, and has clear strong semantics. Atomicity has been adopted as the standard property to have in the development of parallel and distributed systems.

In complement to this work, with Antonella Del Ponzo, Maria Potop-Butucaru, Romaric Ludinard and Sara Tucci we are currently working on the specification of a distributed ledger as the composition of two finite state automata enriched with a consistency criteria that specifies the behavior of the ledger in presence of concurrency. The first automaton, called *blocktree abstract data type*, describes the transition of the tree of blocks of when read and append of new blocks are executed. The second automaton, called *token oracle*, captures the cryptographic process, *proof-of-work*, specific to permissionless ledgers that condition the append of new blocks. Indeed, a particularity of the blockchain technology lies in the notion of *validity* of blocks, i.e. the blockchain must contain only blocks that satisfy a given predicate. As we have shown previously, validity can be achieved through proof-of-work (Dwork and Naor [108]) or other agreement mechanisms. In order to abstract away implementation-specific validation mechanisms, we can encapsulate the validation process in an oracle model separated from the process of updating the data structure. Because the oracle is the only generator of valid blocks and only valid blocks can be appended to the tree, it is the oracle that grants the access to the data structure. The oracle might also own a synchronization power to control the size of forks, i.e., the number of blocks that point back to the same block of the tree. In this respect we may define oracle models such that, depending on the model, the size k of forks can be equal to 1 (i.e., strongest oracle model), strictly greater than 1, or unbounded (i.e., weakest

oracle model).

We have obtained some preliminary results on the implementability of blockchains [41, 42].

- The strongest oracle, guaranteeing no fork, has Consensus number ∞ in the Consensus hierarchy of concurrent objects [137].
- The weakest oracle, which validates a potentially unbounded number of blocks to be appended to a given block, is not stronger than Generalized Agreement Lattice [115].
- The impossibility to guarantee strong prefix in a message-passing system if forks of size $k > 1$ are allowed. This means that Strong Prefix needs the strongest oracle to be implemented, which is at least as strong as Consensus.
- A necessary condition for eventual prefix in a message-passing system is that each update sent by a correct process must be eventually received by every correct process. Informally, eventual prefix says that eventually all read return the same value associated with the greatest score of a chain that is read. Moreover, the result implies that it is impossible to implement eventual prefix if even a single update is dropped at some correct process while it has been received at all the other correct processes.

The proposed framework along with the above-mentioned results should help us in classifying existing blockchains in terms of their consistency and implementability. So far we have shown that Bitcoin [173] and Ethereum [210] have a validation mechanism that maps to our weakest oracle and then they only implement Eventual prefix, while other proposals map to our strongest oracle, falling in the class of those that guarantee Strong Prefix (e.g. Hyperledger Fabric [46], PeerCensus [95], ByzCoin [145]. We are currently revisiting the properties of both the block tree and the token oracle data types to take into account graph-based ledgers. In particular, the presence of forks of size $k > 1$ is different from the one of split chains in Sycomore, while the merged chains notion does not exist in the block tree abstract data type.

2.2 Generalized Lattice Agreement: a new abstraction for constructing distributed ledgers ?

In the Lattice Agreement problem, introduced by Attiya et al. [54], each process p_i has an input value x_i drawn from the join semilattice and must decide an output value y_i, such that *(i)* y_i is the join of x_i and some set of input values and *(ii)* all output values are comparable to each other in the lattice, that is form a chain in the lattice. Lattice Agreement describes situations in which processes need to obtain some knowledge on the global execution of the system, for example a global photography of the system. In particular Attiya et al. [54] have shown that in the asynchronous shared memory computational model, implementing a snapshot object is equivalent to solving the Lattice Agreement problem.

Differently from Consensus, Lattice Agreement can be deterministically solved in an asynchronous setting in presence of crash failures. Faleiro et al. [114] have shown that a majority of correct processes and reliable communication channels are sufficient to solve Lattice Agreement, while Garg et al. [214] proposed a solution that requires $\mathcal{O}(\log n)$ message delays, where n is the number

of processes participating to the algorithm. The very recent solution of Skrzypczak et al. [201] considerably improves Faleiro's construction in terms of memory consumption, at the expense of progress.

In the Generalized Lattice Agreement (GLA) problem processes propose an infinite number of input values (drawn from an infinite semilattice) and decide an infinite sequence of output values, such that, all output values are comparable to each other in the lattice i.e. form a chain (as for Lattice Agreement); the sequence of decision values are non-decreasing, and every input values eventually appears in some decision values. Solving GLA in asynchronous distributed systems reveals to be very powerful as it allows to built a linearizable RSM of commutative update operations [114]. As previously said, replicated satate machine is usually built on top of a Consensus primitive that is used by processes to agree on the order of execution of requests concurrently issued by clients. The main problem with this approach is that consensus is impossible to achieve deterministically in an asynchronous distributed system, and thus this led the research community to study and develop alternative solutions based on the relaxation of some of the constraints, to allow agreement to be reached in partially synchronous systems with faulty processes by trading off consistency with availability. An alternative approach consists in imposing constraints on the set of operations that can be issued by clients, i.e. imposing updates that commute. In particular, in 2012 Faleiro et al. [114] introduced a RSM approach based on a generalized version of the well known Lattice Agreement (LA) problem, that restricts the set of allowed update operations to commuting ones [182]. They have shown that commutative replicated data types (CRDTs) can be implemented with an RSM approach in asynchronous settings using the monotonic growth of a join semilattice, i.e., a partially ordered set that defines a join (least upper bound) for all element pairs. A typical example is the implementation of a dependable counter with `add` and `read` operations, where updates (i.e. `add`s) are commutative.

Despite recent advancements in this field, to the best of our knowledge no general solution exists that solves Lattice Agreement problems in an asynchronous setting with Byzantine faults.

With Querzoni and Di Luna [161], we are currently continuing the line of research on Lattice Agreement in asynchronous message-passing systems by considering a Byzantine fault model. Briefly, processes communicate by exchanging messages over asynchronous authenticated reliable point-to-point communication links (messages are never lost on links, but delays are unbounded). The communication graph is complete: there is a communication link between each pair of processes. We have a set $F \subset P$ of Byzantine processes, with $|F| \leq f$. Byzantine processes deviate arbitrarily from the algorithm. We have shown that $|P| \geq 3f + 1$ is necessary. In the Lattice Agreement problem, each process $p_i \in C$ starts with an initial input value $pro_i \in V$. Values in V form a join semi-lattice $L = (V, \oplus)$ for some commutative join operation $\oplus$: for each $u, v \in V$ we have $u \leq v$ if and only if $v = u \oplus v$. Given $V' = \{v_1, v_2, \ldots, v_k\} \subseteq V$ we have $\bigoplus V' = v_1 \oplus v_2 \oplus \ldots \oplus v_k$. We formalize the Lattice Agreement in a Byzantine context by the following properties:

- **Liveness:** Each process $p_i \in C$ eventually outputs a decision value $dec_i \in V$;
- **Stability:** Each process $p_i \in C$ outputs a unique decision value $dec_i \in V$;
- **Comparability:** Given any two pair $p_i, p_j \in C$ we have that either $dec_i \leq dec_j$ or $dec_j \leq dec_i$;
- **Inclusivity:** Given any correct process $p_i \in C$ we have that $pro_i \leq dec_i$;

- **Non-Triviality:** Given any correct process $p_i \in C$ we have that $dec_i \leq \bigoplus(X \cup B)$, where X is the set of proposed values of all correct processes ($X : \{pro_i|$ with $p_i \in C\}$), and $B \subset V$ is $|B| \leq f$.

We have proposed an algorithm, namely *Wait Till Safe* (WTS), which, in presence of less than $(n-1)/3$ Byzantine processes, guarantees that any correct process decides in no more than $5+2f$ message delays with a global message complexity in $\mathcal{O}(n^2)$ per process. We show that $(n-1)/3$ is an upper bound. Our algorithm makes use of a Byzantine reliable broadcast primitive to circumvent adversarial runs where a Byzantine process may induce correct processes to deliver different input values. A Byzantine reliable broadcast communication primitive combined with a "wait until safe" strategy ensures that adversarial scenarios in which Byzantine processes pretend to have a non comparable state with correct processes is circumvented. The algorithm is wait-free, i.e., every process completes its execution of the algorithm within a bounded number of steps, regardless of the execution of other processes.

We have then proposed an algorithm, namely *Generalized Wait Till Safe* (GWTS), to solve GLA in a Byzantine fault model. Here the challenge is twofold: first, we need to guarantee that, despite the fact that input values are proposed in tumbling batches, Byzantine processes cannot keep rejecting all new proposals under the pretext that they are not comparable with the current ones. Second, we must ensure that adversarial processes cannot progress much faster than all the other correct processes (i.e., output decision values faster than correct processes) which would allow them to prevent all correct processing from proposing their own values. Our "wait until safe" strategy guarantees that each correct process performs an infinite sequence of decisions, and for each input received at a correct process, its value is eventually included in a decision. Our algorithm is wait-free and is resilient to $f \leq (n-1)/3$ Byzantine processes.

Based on *Generalized Wait Till Safe* (GWTS), we have proposed the construction of a RSM for objects with commuting update operations that guarantees both linearizability and progress in asynchronous environments with Byzantine failures. Our construction is resilient to f Byzantine replicas, and to any number of Byzantine clients. Actually we are not aware of any similar results.

Based on these results, we would like to further explore the relationships between generalized lattice agreement and distributed ledgers. The reasons why both problems seem to be strongly related is that fact that in both problems we aim at creating comparable decision values. However, in GLA decision values are set of elements of a (infinite) lattice while in distributed ledgers, decision values are sequence of elements.

2.3 Performance Issues of Distributed Ledgers

2.3.1 Toward Permissionless Blockchains with an Adaptive Throughput

Bitcoin blockchain, and more generally cryptocurrency systems have shown that in a highly adversarial environment where mutually distrustful nodes are participating, robustness is achievable. But this is at the expense of performance. This compels permissionless blockchains to a low transaction confirmation throughput. For instance, in Bitcoin no more than 7 transactions can be permanently

confirmed per second in average. In Ethereum this raises up to 20 per second. Still this is by too far very low.

A recent evolution in the structure of blockchains is emerging to deal with the performance issue aforementioned. Eyal et al. [112] propose with Bitcoin-NG an off-chain mechanism: blocks refer to a leader in charge of validating transactions batched in micro-blocks out of the chain. Lightning [179] follows the same principle by publishing results of a set of transactions among two parties. HashGraph [60], ByteBall [89], and Iota [180] leverage the presence of well known institutions to get rid of blocks, Ghost [203] and Spectre [202] protocols family modifies the blockchain data structure from linked-list of blocks to a directed graph of blocks. In particular, Spectre [202] organises blocks in a directed (but not acyclic) graph of blocks. Blocks are built so that they acknowledge the state of the directed graph at the time blocks were created which decreases the opportunity for powerful attackers to create blocks in advance. The number of created blocks is increased with respect to classic blockchain-based systems, since all the concurrently created blocks belong to the directed graph. Unfortunately the absence of mechanisms to prevent the presence of conflictual records (i.e., blocks with conflicting transactions) and the cycles in the directed graph require that participants execute a complex algorithm to extract from the graph the set of accepted (i.e., valid) transactions [202].

The point is that totally ordering blocks is simply not necessary. It makes easier the validation process of blocks, but relying on a single chain compels the creation of blocks to be (almost) fixed whatever the flow of transactions issued by clients. With Antoine Guellier and Romaric Ludinard we have explored how to improve upon this issue by relaxing the total order to a simpler partial order [29]. Revisiting the distributed ledger structure by organizing it as a partially ordered set of blocks that better reflects the natural ordering over blocks has led to Sycomore. Sycomore is a ledger in which blocks are organized within a dedicated directed acyclic graph, called syc-dag. Sycomore resembles in many aspects to Nakamoto blockchain. The same four ingredients are used i.e., the UTXO model, a randomized election implemented through a proof-of-work (PoW), a Fork Choice Rule (FCR) strategy, and a short latency broadcast primitive. On the other hand, the unique features of Sycomore are the following ones.

1. Sycomore is a ledger whose structure is a directed acyclic graph (syc-dag) of blocks
2. The structure of syc-dag self-adapts to fluctuations of transactions demand: branchs of blocks in syc-dag self split into multiple ones to cope with burst of pending transactions, while they self merge back together to adapt to a drop of activity;
3. The frequency at which blocks are mined self-adapts to the number of leaf-blocks in syc-dag;
4. The probability of forks decreases with the increasing number of leaf blocks. As a consequence miners can mine in parallel without incurring more forks than in Bitcoin
5. Miners cannot predict the branch in syc-dag to which their blocks will be appended. This prevents malicious miners from devoting their computational work on the growing of specific branches of the syc-dag.
6. Transactions are dynamically and evenly partitioned over the syc-dag, i.e., there is no transaction that belongs to two different blocks. This makes effective the parallelism brought by the syc-dag, which allows us to increase the number of confirmed transactions per second.

Altogether, these properties should allow us to build a secure, immutable and efficient distributed ledger whose structure self-adapts to transactions demand. During the next months, our objective is to formally prove the correctness of Sycomore by using the interactive theorem prover Coq. The challenge that we will have to face is to take into account the random nature of proof-of-work. The ultimate objective being to propose Sycomore to Bitcoin developers.

2.3.2 Toward Sustainable Permissionless Blockchains

Resilience of PoW-based solutions fundamentally relies on the massive use of computational resources (in particular from correct nodes that must devote as much as computations as the (adaptive) adversary). This is a real issue today. Lot of investigations have been devoted to find a secure alternative to PoW, but most of them either rely on the intensive use of a large quantity of physical resources (e.g., proof-of-space [52], proof-of-space/time [168]) or makes compromises in their trust assumptions (e.g., proof-of-elapsed-time [53], delegated proof-of-stake [110]). In contrast, solutions based on proof-of-stake (PoS) seem to be a quite promising way to build secure and permissionless blockchains. Indeed, proof-of-stake rely on a limited but abstract resource, the crypto-currency, in such a way that the probability for a participant to create the next block of the blockchain is generally proportional to the fraction of currency owned by this participant. It is an elegant alternative in the sense that all the information needed to verify the legitimacy of a stakeholder to create a block (i.e., crypto-currency possession) is already stored in the blockchain. Finally, by being a sustainable alternative (creating a block requires a few number of operations), scalability concerns, exhibited by PoW-based solutions, should be a priori more tractable.

An important condition for a PoS-blockchain to be secure is randomness. The creator of the next block must be truly random, and the source of randomness must not be biaised by any adversarial strategy. So far, this has been achieved by two main approaches: chain-based consensus and block-wise Byzantine agreement. Ouroboros [143], representative of the chain-based approach, is a synchronous PoS protocol resilient against a weakly dynamic adversary that owns $1/2 - \epsilon$ of stake. A snapshot of the current users' status is periodically taken, from which the the next sequence of leaders is computed. Note that Ouroboros has been recently improved to work in the partially synchronous setting against a dynamic adversary [59, 65], but keeping the same design principles as the original one. In Ouroboros, a unique leader is elected at each round to broadcast its block which contrasts with our sharded approach where the block creation process is distributed. Algorand [128], is a representative of the blockwise Byzantine agreement approach. It provides a distributed ledger against an strongly adaptive adversary by relying on the properties of verifiable random cryptographic schemes without assuming strong synchrony assumptions. However, by its design, agreement for each block of the blockchain is achieved by involving a very large number of stakeholders so that each one needs to effectively participate only for one exchange of messages.

With Antoine Durand and Romaric Ludinard [105], we have recently proposed a new blockchain protocol called StakeCube which aims at improving scalability of the block-wise Byzantine agreement approach by combining sharding techniques, users presence and stake transfer to operate in a PoS setting.

The adversary model is the "Stake Threshold Adversary" by Abraham and Malkhi [2], an extension

(or modification) of the Computational Threshold Adversary, which bounds the proportion of the stake owned by parties. Specifically, the adversary controls up to $\mu \leq 1/3$ of the total amount of stake currently available in the system. The adversary is weakly adaptive, in the sense that it can select at any time which parties to corrupt in replacement of corrupted ones (i.e., corruptions are "moving"), however a corruption becomes effective T blocks after the adversary has selected the party to be corrupted.

The key idea of StakeCube is to organise users (i.e., stakeholders) into shards— such that the number of shards increases sub-linearly with the total number of active UTXOs— and within each shard, to randomly choose a constant size committee in charge of executing the distributed algorithms that contribute to the creation of blocks. Each block at height h in the blockchain is by design unique (no fork), and once a block is accepted in the blockchain, the next one is created by a sub-committee of shards whose selection is random with a distribution that depends on the content of the last accepted block. In StakeCube, participation of honest users is conditional to the possession of UTXOs. Participation is voluntary: Any honest user can join a shard (determined by the protocol), whenever she wishes, with the objective of eventually being involved in the Byzantine resilient protocols executed in this shard. Participation is temporary: The sojourn time of an honest user in a shard is defined by the time it takes for StakeCube to create T blocks. Once she leaves, she can participate again by joining another shard, and does so until she spends her UTXO. As users may own multiple UTXOs, they can simultaneously and verifiably sit in different shards. As in PeerCube, we rely on induced churn to protect StakeCube against the weakly adaptive adversary. However, Peercube critically relies on a (global) trusted party supplying verifiable random identifiers to nodes (i.e., a PKI). in StakeCube we take advantage of the already known public keys (of the UTXO model) and some randomness present in each block to build verifiable random identifiers. Then to cope with this induced churn, shards' views are updated, signed and installed once, and this occurs right before the acceptation of a new block. Finally, the creation of blocks is efficiently handled by an agreement among a verifiable sub-committee of shards. We might expect that solely relying on stakeholders (i.e., owners of the coins of the crypto-currency system) to the secure construction of the blockchain makes sense due to their incentive to be fully involved in the blockchain governance, rather than delegating it to powerful miners. Analysis against rational players is an objective I intend to address in the future.

2.4 A permissionless Supply Chain Application

The food supply chain is one of the most complex and fragmented of all supply chains. Production is found all over the world, both on land and in water, making it difficult to identify and track many producers and intermediaries. The process of monitoring the quality of crops (right from harvest to delivery) has never been easy. It's basically a huge challenge for farmers and growers throughout the world. IoT makes possible for growers to monitor soil quality, and irrigation in a precise and highly efficient manner. It is even more important for organic food which require very fast delivery. I am convinced that the blockchain technology has the capability to improve traceability of crops and deliver better outcomes. Essentially, with a blockchain ledger, we should get to know the status of crops right from planting to delivery, the presence of grain certificates in a permanent and secure way. The blockchain should allow to improve the frequency of data verification, its real-time access, and its immutability. Developing such an application in a permissionless context is an exciting

objective I intend to study in the near future.

Bibliography

[1] M. Abboud, C. Delporte-Gallet, and H. Fauconnier. Agreement and consistency without knowing the number of correct processes. *Proceedings of International Conference on New Technologies of Distributed Systems (Notère))*, 2008.

[2] I. Abraham and D. Malkhi. The blockchain consensus layer and BFT. *Bulletin of the EATCS*, 3(123):1–23, 2017.

[3] H. Acan, A. Collevecchio, A. Mehrabian, and W. Nick. On the push&pull protocol for rumour spreading. In *Proceedings of the ACM Symposium on Principles of Distributed Systems (PODC)*, 2015.

[4] E. Adar and B. Huberman. Free-riding on gnutella. Online at http://www.firstmonday.org/issues/issue5_10/adar/index.html, 2000.

[5] M. K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, 2004.

[6] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1999.

[7] B. Alpern and F. B.Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[8] E. Anceaume. A comparison of atomic broadcast protocols in time-bounded distributed systems. In *Proceedings of the Fourth Worshop on Future Trends (FTDCS), Distributed Computing Systems*, pages 166–172, Lisbonne, Portugal, 1993. IEEE.

[9] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 292–301, June 1997.

[10] E. Anceaume. An efficient solution to uniform atomic broadcast. *International Journal of Foundations of Computer Science*, 13(5):695–717, 2002.

[11] E. Anceaume, F. Brasileiro, R. Ludinard, and A. Ravoaja. Peercube: An hypercube -based p2p overlay robust against collusion and churn. In *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2008.

[12] E. Anceaume, F. Brasileiro, R. Ludinard, B. Sericola, and F. Tronel. Analytical study of adversarial strategies in cluster-based overlays. In *Proceedings of the International Workshop on on Reliability, Availability, and Security (WRAS)*, 2009.

[13] E. Anceaume, F. Brasileiro, R. Ludinard, B. Sericola, and F. Tronel. Brief announcement: Induced churn to face adversarial behavior in peer-to-peer systems. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2009.

[14] E. Anceaume, F. Brasileiro, R. Ludinard, B. Sericola, and F. Tronel. Modeling and evaluating targeted attacks in dynamic systems. In *Proceedings of the 41th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2011.

[15] E. Anceaume, Y. Busnel, and S. Gambs. Uniform and ergodic sampling in unstructured peer-to-peer systems with malicious nodes. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems (OPODIS)*, 2010.

[16] E. Anceaume, Y. Busnel, and S. Gambs. On the power of the adversary to solve the node sampling problem. *Transactions on Large-Scale Data and Knowledge-Centered Systems*, pages 102–126, 2013.

[17] E. Anceaume, Y. Busnel, E. Le Merrer, R. Ludinard, M. Jean-Louis, and B. Sericola. Anomaly Characterization in Large Scale Networks. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[18] E. Anceaume, Y. Busnel, and B. Sericola. Uniform node sampling service robust against collusions of malicious nodes. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[19] E. Anceaume, F. Castella, R. Ludinard, and B. Sericola. Markov chains competing for transitions: Application to large-scale distributed systems. *Journal of Methodology and Computing in Applied Probability*, 15(2):305–332, 2013.

[20] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report 95-1534, Cornell University, Ithaca, New York, 1995.

[21] E. Anceaume, A. Datta, M. Gradinariu, and A. Virgillito. Semantic overlay for self-* peer-to-peer publish/subscribe. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2006.

[22] E. Anceaume, X. Defago, M. Gradinariu, , and M. Roy. Toward a theory of self-organization. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[23] E. Anceaume, X. Defago, M. Gradinariu, and M. Roy. Toward a theory of self-organization: Case study. In *Proceedings of the International Symposium on DIStributed Computing (DISC), short paper*, 2005.

[24] E. Anceaume, C. Delporte-Gallet, H. Fauconnier, and G. L. Lann. Designing modular services in the scattered byzantine failure model. In *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC)*, 2004.

[25] E. Anceaume, C. Delporte-Gallet, H. Fauconnier, and G. L. Lann. Modular services with byzantine recovery. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2004.

[26] E. Anceaume, C. Delporte-Gallet, H. Fauconnier, and M. H. J. Widder. Clock synchronization in the byzantine-recovery failure model. In *Proceedings of the Conference on Principles of Distributed Systems (OPODIS)*, 2007.

[27] E. Anceaume, R. Friedman, and M. Gradinariu. Managed agreement: Generalizing two fundamental distributed agreement problems. *Information Processing Letters*, 101(5):190–198, 2007.

[28] E. Anceaume, M. Gradinariu, and A. Ravoaja. Incentive for p2p fair resource sharing. In *Proceedings of the IEEE P2P Conference (P2P)*. IEEE, 2005.

[29] E. Anceaume, A. Guellier, R. Ludinard, and B. Sericola. Sycomore: A permissionless distributed ledger that self-adapts to transaction demand. In *Proceedings of the 17th IEEE International Conference on Network Computing and Applications (NCA)*, 2018.

[30] E. Anceaume, G. Guette, P. Lajoie-Mazenc, T. Sirvent, and V. V. T. Tong. Extending signatures of reputation. In *Proceedings of the 8th International IFIP Summer School on Privacy and Identity Management for Emerging Services and Technologies*, 2013.

[31] E. Anceaume, G. Guette, P. Lajoie-Mazenc, T. Sirvent, and V. V. T. Tong. Extending signatures of reputation. *IFIP Advances in Information and Communication Technology*, pages 165–176, 2014.

[32] E. Anceaume, J.-M. Hélary, and M. Raynal. Tracking immediate predecessors in distributed computations. In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures (SPAA'02)*. ACM, 2002.

[33] E. Anceaume, T. Lajoie-Mazenc, R. Ludinard, and B. Sericola. Safety Analysis of Bitcoin Improvement Proposals. In *IEEE Symposium on Network Computing and Applications (NCA)*, 2016.

[34] E. Anceaume, E. Le Merrer, R. Ludinard, B. Sericola, and G. Straub. A Self-organising Isolated Anomaly Detection Architecture for Large Scale Systems. In *Proceedings of 2013 NEM Summit conference*, 2013.

[35] E. Anceaume, R. Ludinard, M. Potop-Butucaru, and F. Tronel. Bitcoin a Distributed Shared Register. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 456–468, 2017.

[36] E. Anceaume, R. Ludinard, and B. Sericola. Analytical study of the impact of churn in a cluster-based structured P2p overlay. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2010.

[37] E. Anceaume, R. Ludinard, and B. Sericola. Performance evaluation of large-scale dynamic systems. *ACM SIGMETRICS Performance Evaluation Review, Special Issue on Modeling Dynamic Behaviors of Complex Distributed Systems*, 39(4):108–117, 2012.

[38] E. Anceaume, E. L. Merrer, R. Ludinard, B. Sericola, and G. Straub. Fixme: A self-organizing isolated anomaly detection architecture for large scale distributed systems. In *Proceedings of the 16th International Conference On Principles Of Distributed Systems (OPODIS)*, 2012.

[39] E. Anceaume and P. Minet. Analysis of atomic or causal broadcast protocols. *RIR : Revue Réseaux et Informatique Répartie*, 1(3):257–274, 1991.

[40] E. Anceaume and E. Mourgaya. Unreliable distributed timing scrutinizer: Adapting asynchronous algorithms to the environment. In *Proceedings of the 5th International Symposium*

on Object-Oriented Real-Time Distributed Computing (ISORC 2002), Washington DC, USA, avril 2002. IEEE.

[41] E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci-Piergiovanni. Blockchain abstract data type. In *Proceedings of the 24th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019. Brief.

[42] E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, and S. Tucci-Piergiovanni. Blockchain abstract data type. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

[43] E. Anceaume and I. Puaut. A taxonomy of clock synchronization algorithms. Technical Report 1103, IRISA, July 1997.

[44] E. Anceaume and I. Puaut. Performance evaluation of clock synchronization algorithms. Technical Report 1208, IRISA, Oct. 1998.

[45] E. Anceaume and A. Ravoaja. Incentive-based robust reputation mechanism for p2p services. In *Proceedings of the 10th International Conference On Principles Of Distributed Systems (OPODIS)*, 2006.

[46] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. https://arxiv.org/pdf/1801.10228v1.pdf, 2018.

[47] E. Androulaki, S. Choi, S. Bellovin, and T. Malkin. Reputation systems for anonymous networks. In *Privacy Enhancing Technologies*, pages 202–218. Springer, 2008.

[48] D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(2):183–199, 2008.

[49] A. F. Anta, C. Georgiou, and N. C. Nicolaou. Atomic appends: Selling cars and coordinating armies with multiple distributed ledgers. *Tokenomics 2019*, abs/1812.08446, 2018.

[50] A. F. Anta, K. Konwar, C. Georgiou, and N. Nicolaou. Formalizing and implementing distributed ledger objects. *ACM SIGACT News*, 49(2):58–76, 2018.

[51] K. Antoniadis, R. Guerraoui, D. Malkhi, and D. Seredinschi. State machine replication is more expensive than consensus, 2018.

[52] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of Space: When Space Is of the Essence. In *Proceedings of the International Conference on Security and Cryptography for Networks (SCN)*, 2014.

[53] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of Space: When Space Is of the Essence. In *Proceedings of the International Conference on Security and Cryptography for Networks (SCN)*, 2014.

[54] H. Attiya, M. Herlihy, and O. Rachman. Atomic snaphots using lattice agreement. *Distributed Computing*, 8:121–132, 1995.

[55] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience publishers, 2004.

[56] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[57] B. Awerbuch and C. Scheideler. Towards a scalable and robust dht. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2006.

[58] B. Awerbuch and C. Scheideler. Towards scalable and robust overay networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[59] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[60] L. Baird. The SWIRLDS Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance. http://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf, 2016.

[61] R. Baldoni, M. Bertier, M. Raynal, and S. T. Piergiovanni. Looking for a definition of dynamic distributed systems. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14, 2007.

[62] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Proc. of ICDCS 2005*, 2005.

[63] R. F. Baumeister, E. Bratslavsky, C. Finkenauer, and K. D. Vohs. Bad is stronger than good. *Review of general psychology*, 5(4):323, 2001.

[64] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1983.

[65] D. Bernardo, P. Gaži, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *International Conference on the Theory and Applications of Cryptographic (EUROCRYPT)*, 2018.

[66] P. Bernstein, V. Hadzilacos, and H. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[67] A. N. Bessani and E. Alchieri. A guided tour on the theory and practice of state machine replication. http://www.di.fc.ul.pt/~bessani/publications/tutorial-smr.pdf, 2014.

[68] J. Bethencourt, E. Shi, and D. Song. Signatures of reputation: Towards trust without identity. *Financial Cryptography and Data Security*, pages 400–407, 2010.

[69] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2015.

[70] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer. Brahms: Byzantine Resilient Random Membership Sampling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2008.

[71] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[72] S. Buchegger and J. L. Boudec. A robust reputation system for p2p and mobile ad-hoc networks. In *Proceedings of the Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[73] C. Cachin. Blockchain - From the Anarchy of Cryptocurrencies to the Enterprise (Keynote Abstract). In *Proceedings of the OPODIS International Conference*, 2016.

[74] C. Cachin, R. Guerraoui, and L.Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, 2nd edition, 2011.

[75] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO*, 2001.

[76] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.

[77] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[78] M. Castro, P. Druschel, A. M. Kermarrec, and A. Rowston. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[79] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[80] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[81] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[82] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[83] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4), 1996.

[84] B. Charron-Bost and A. Schiper. Harmful dogmas in fault tolerant distributed computing. *SIGACT News*, 38(1), 2007.

[85] B. Charron-Bost and A. Schiper. The heard-of model: Computing in distributed systems with benign faults. *Distributed Computing*, 22(1), 2009.

[86] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonums. *Communications of the ACM*, 24(2):84–90, 1981.

[87] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[88] F. Chu. Reducing ω to $\diamond\sqsupseteq$. *Information Processing Letters*, 67:289–293, 1998.

[89] A. Churyumov. ByteBall : A Decentralized System for Storage and Transfer of Value. https://byteball.org/Byteball.pdf, 2017.

[90] S. Clauß, S. Schiffner, and F. Kerschbaum. k-anonymous reputation. In K. Chen, Q. Xie, W. Qiu, N. Li, and W.-G. Tzeng, editors, *ASIACCS*, pages 359–368. ACM, 2013.

[91] T. Condie, V. Kacholia, S. Sank, J. Hellerstein, and P. Maniatis. Induced Churn as Shelter from Routing-Table Poisoning. In *Proceedings of the International Network and Distributed System Security Symposium (NDSS)*, 2006.

[92] M. Correia, G.-S. Veronese, N.-F. Neves, and P. Verissimo. Byzantine consensus in asynchronous message passing systems; a survey. *International Journal of Critical Computer-Based Systems*, 2(2), 2011.

[93] D. Daley and D. G. Kendall. Stochastic rumours. *IMA Journal of Applied Mathematics*, 1(1):42–55, 1965.

[94] C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *17th International Conference on Distributed Computing and Networking (ICDCN)*, 2016.

[95] C. Decker, J. Seidel, and R. Wattenhofer. Bitcoin Meets Strong Consistency. In *Proc. of the ICDCN International Conference*, 2016.

[96] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[97] C. Dellarocas. Reputation mechanisms. In T. Hendershott, editor, *Handbook on Information Systems and Economics*. Elsevier Publishing, 2006.

[98] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 338–346, 2004.

[99] A. Demers, M. Gealy, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated datbase maintenance. In *Proceedings of the ACM Syposium on Principles of Distributed Systems (PODC)*, 1987.

[100] Z. Despotovic. *Building trust-aware P2P systems : from trust and reputation management to decentralized e-commerce applications*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2005.

[101] J. C. Diehl C. and R. J.-X. Reachability analysis of distributed executions. In *Proceedings of Theory and Practice of Software Development Conferance (TAPSOFT)*, 1993.

[102] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1), Jan. 1987.

[103] D. Dolev and R. Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

[104] J. Douceur and J. S. Donath. The Sybil Attack. In *Proceedings of the 1rst International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260, 2002.

[105] A. Durand, E. Anceaume, and R. Ludinard. Stakecube: Combining sharding and proof-of-stake to build fork-free secure permissionless distributed ledgers. In *Proceedings of the International conference on networked systems (NETYS)*, 2019.

[106] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 1988.

[107] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[108] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proceedings of the Annual International Cryptology Conference - Advances in Cryptology (CRYPTO)*, 1992.

[109] E.Anceaume, E. Mourgaya, and P. R. Parvédy. Unreliable distributed timing scrutinizer to converge toward decision conditions. *Studia Informatica Universalis*, 3(1):19–42, 2004.

[110] EOS.IO. Technical white paper v2, 2019. Accessed: 2019-03-10.

[111] European Parliament and Council of the European Union. Directive 95/46/EC, 1995.

[112] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[113] M. F. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1988.

[114] J. Faleiro, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *Proceedings of the ACM Principles of Distributed Computing (PODC)*, 2012.

[115] J. M. Falerio, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2012.

[116] P. Felber and R. Chand. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proc. of EUROPAR 2005*, 2005.

[117] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 427–438. IEEE, 1987.

[118] A. Fiat, J. Saia, and M. Young. Making chord robust to byzantine attacks. In *Proceedings of the Annual European Symposium on Algorithms (AESA)*, 2005.

[119] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[120] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[121] M. J. Fischer and M. Merritt. Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2–3):239–247, 2003.

[122] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, 2005.

[123] A. Frieze and G. Grimmet. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57–77, 1985.

[124] A. J. Ganesh. Rumour spreading on graphs. Technical report, 2015.

[125] J. Garay and A. Kiayias. Sok: A consensus taxonomy in the blockchain era. Cryptology ePrint Archive, 2019. https://eprint.iacr.org/2018/754.

[126] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proc. of the EUROCRYPT International Conference*, 2015.

[127] V. K. Garg and N. Mittal. Time and state in asynchronous distributed systems. In *Encyclopedia of Computer Science and Engineering*. Wiley, 2008.

[128] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Symposium on Operating Systems Principles (SOSP)*, 2017.

[129] A. Girault, G. Gössler, R. Guerraoui, J. Hamza, and D.-A. Seredinschi. Monotonic prefix consistency in distributed systems. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 41–57, Berlin, Germany, 2018. Springer.

[130] P. B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proceedings of the Annual ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2006.

[131] R. Guerraoui. Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors. *Distributed Computing*, 15(1):17–25, 2002.

[132] W. guilera, M. K.and Chen and S. Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, 2000.

[133] A. Gupta, O. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish:subscribe over p2p networks. In *Proc. of IFIP/ACM Middleware'04*, 2004.

[134] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.

[135] O. Hasan, L. Brunie, and E. Bertino. Preserving privacy of feedback providers in decentralized reputation systems. *Computers & Security*, 31(7):816–826, 2012.

[136] M. Herlihy. Concurrency and availability as dual properties of replicated atomic data. *J. ACM*, 37(2):257–278, 1990.

[137] M. Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[138] M. Herlihy and M. Moir. Blockchains and the logic of accountability: Keynote address. In *Proceedings of the ACM/IEEE LICS Symposium*, 2016.

[139] K. Hildrum and J. Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proc. of the 17th Int'l Symposium on Distributed Computing (DISC)*, 2003.

[140] J. Hoepman, M. Papatriantafilou, and P. Tsigas. Self-stabilization of wait-free shared memory objects. *J. Parallel Distrib. Comput.*, 62(5):818–842, 2002.

[141] G. P. Jesi, A. Montresor, and M. van Steen. Secure Peer Sampling. *Computer Networks*, 54(12):2086–2098, 2010.

[142] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 32(2), 2001.

[143] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. Cryptology ePrint Archive, Report 2016/889, 2016. https://eprint.iacr.org/2016/889.

[144] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium*, 2016.

[145] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2016.

[146] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):1–39, 2009.

[147] K. Kursawe. Optimistic byzantine agreement. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2002.

[148] P. Lajoie-Mazenc, E. Anceaume, G. Guette, T. Sirvent, and V. V. T. Tong. Mécanisme de réputation distribué préservant la vie privée avec témoignages négatifs. In *Actes des 17èmes Rencontres Francophones pour les Aspects Algorithmiques des Télécommunications (Algotel)*, 2015.

[149] P. Lajoie-Mazenc, E. Anceaume, G. Guette, T. Sirvent, and V. V. T. Tong. Privacy-preserving reputation mechanism: A usable solution handling negative ratings. In *Proceedings of the 9th IFIP WG 11.11 International Conference on Trust Management (IFIP TM)*, 2015.

[150] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 1(2):95–114, 1978.

[151] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[152] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, pages 254–280, 1984.

[153] L. Lamport. On inter-process communications, part I: basic formalism and part II: algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[154] L. Lamport. The Part-time Parliament. *ACM Transaction on Computer Systems*, 16(2), 1998.

[155] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[156] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[157] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1985.

[158] D. Liu, P. Ning, and W. Du. Detecting Malicious Beacon Nodes for Secure Location Discovery in Wireless Sensor Networks. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[159] T. Locher, S. Schmid, and R. Wattenhofer. eQuus: A provably robust and locality-aware peer-to-peer system. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*, 2006.

[160] M. Luby. LT codes. In *Proceedings of the IEEE International Symposium on Foundations of Computer Science (SFCS)*, 2002.

[161] G. A. D. Luna, E. Anceaume, and L. Querzoni. Byzantine generalized lattice agreement. In *Proceedings of the 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2020.

[162] N. Lynch. *Distributed algorithms*. Morgan Kaufmann publishers, 1996.

[163] J. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3), 2006.

[164] P. Maymounkov. Online codes. *Research Report TR2002-833, New York University*, 2002.

[165] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[166] Y. Mocquard, B. Sericola, and E. Anceaume. Probabilistic analysis of rumor spreading. *Informs Journal on Computing*, pages 1–24, 2019.

[167] Y. Mocquard, B. Sericola, S. Robert, and E. Anceaume. Analysis of the Propagation Time of a Rumour in Large-scale Distributed Systems. In *Proceedings of the IEEE International Conference on Network Computing and Applications (NCA)*, 2016. This article has received the Best Student Paper Award.

[168] T. Moran and I. Orlov. Proofs of space-time and rational proofs of storage. In *Cryptology ePrint Archive*, 2016.

[169] A. Mostéfaoui, E. Mourgaya, P. R. Parvédy, and M. Raynal. Evaluating the condition-based approach to solve consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2003.

[170] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2003.

[171] A. Mostéfaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. E. Abbadi. From static distributed systems to dynamic systems. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 109–118, 2005.

[172] A. Mostéfaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. El Abbadi. From static distributed systems to dynamic systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2005.

[173] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[174] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[175] K. Panagiotou and L. Speidel. Asynchronous rumor spreading on random graphs. *Algorithmica*, 78(3):968—-989, 2017.

[176] R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques - Advances in Cryptology - (EUROCRYPT)*, 2017.

[177] E. Pavlov, J. Rosenschein, and Z. Topol. Supporting privacy in decentralized additive reputation systems. *Trust Management*, pages 108–119, 2004.

[178] L. Pease, P. Shostak, and L. Lamport. Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

[179] J. Poon and T. Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf, 2016.

[180] S. Popov. The Tangle. https://iota.org/IOTA_Whitepaper.pdf, 2017.

[181] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1992.

[182] N. M. Preguiça, C. Baquero, and M. Shapiro. Conflict-free replicated data types crdts. In *Encyclopedia of Big Data Technologies*. 2019.

[183] M. O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*, 1983.

[184] P. Ramanathan, K.-G. Shin, and R.-W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10), 1990.

[185] A. Ravoaja and E. Anceaume. Storm: A secure overlay for p2p reputation management. In *Proceedings of the International Conference on Self-Autonomous and Self-Organizing Systems (SASO)*, 2007.

[186] H. Ribeiro and E. Anceaume. A comparative study of rateless codes for P2P distributed storage. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2010.

[187] H. Ribeiro and E. Anceaume. DataCube: a P2P persistence storage architecture based on hybrid redundancy schema. In *Proceedings of the International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, 2010.

[188] H. Ribeiro and E. Anceaume. Exploiting rateless coding in structured overlays to achieve data persistence. In *Proceedings of the IEEE 24th International Conference on Advanced Information Networking and Application (AINA)*, 2010.

[189] L. Rodrigues, R. Baldoni, E. Anceaume, and M. Raynal. Deadline-constrained causal order. In *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'2K)*, May 2000.

[190] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[191] M. Rosenfeld. Analysis of hashrate-based double spending. http://arxiv.org/abs/1402.2009, 2014.

[192] A. Schiper and S. Toueg. From set membership to group membership: A separation of concerns. *IEEE Transactions on Dependable and Secure Computing*, 3(1):2–12, 2006.

[193] F. B. Schneider. A paradigm for reliable clock synchronization. Technical report, Ithaca, NY, USA, 1986.

[194] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[195] J. Shneidman and D. Parkes. Rationality and self-interest in peer to peer networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, 2003.

[196] A. Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking*, pages 2551–2567, 2006.

[197] B. Simons, J. L. Welch, and N. A. Lynch. An overview of clock synchronization. In *Proceedings of the Asilomar Fault-Tolerant Distributed Computing Workshop*, 1986.

[198] A. Singh, T. Ngan, P. Drushel, and D. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, 2006.

[199] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[200] D. Skeen. Crash Recovery in a Distributed Database System. Memorandum No. UCB/ERL M82/45, Electronics Research Laboratory, Berkeley, 1982.

[201] J. Skrzypczak, F. Schintke, and T. Schuütt. Brief announcement: Linearizable state machine replication of state-based crdts without logs. In *Proceedings of the ACM Principles of Distributed Computing (PODC)*, 2019.

[202] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016, 2016.

[203] Y. Sompolinsky and A. Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. *IACR Cryptology ePrint Archive*, 2013:881, 2013.

[204] Y. J. Song and R. Van Renesse. Bosco: One-step byzantine asynchronous consensus. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2008.

[205] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34:626–645, 1987.

[206] S. Steinbrecher. Enhancing multilateral security in and by reputation systems. *The Future of Identity in the Information Society*, pages 135–150, 2009.

[207] P. Triantafillou and I. Aekaterinidis. Content-based Publish/Subscribe over Structured P2P Networks. In *Proc. of DEBS 2004*, 2004.

[208] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *n Proceedings of the International Conference on Dependable Systems and Networks (DSN), p*, 2003.

[209] J. Widder and U. Schmid. The theta-model: achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, 2009.

[210] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf.

[211] B. Yu and M. Singh. Detecting deception in reputation management. In *Proceedings of the ACM-SIGART International joint Conference on Autonomous agents and multiagent systems (AAMAS)*, 2003.

[212] B. Yu, M. P. Singh, and K. Sycara. Developing trust in large-scale peer-to-peer systems. In *Proceedings of IEEE Symposium on Multi-Agent Security and Survivability*, 2004.

[213] G. Zacharia. Trust management through reputation mechanisms. In *Proceedings of the International Conference on Autonomous Agents (Workshop on Deception, Fraud and Trust)*, 1999.

[214] X. Zheng, C. Hu, and V. K. Garg. Lattice agreement in message passing systems. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2018.

[215] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of the Int. Workshop on Network and OS Support for Digital Audio and Video*, 2001.