Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems

Victorien Elvinger, Gérald Oster and François Charoy
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
Email: victorien.elvinger@loria.fr

***Abstract*—In distributed collaborative systems, participants maintain a replicated copy of shared documents. They edit their own copy and then share their modifications without any coordination. Copies follow successions of divergence and convergence. Convergence is a liveness property of collaborative systems. Some malicious participants may find an advantage to make the collaboration fail. To that end, they can preclude convergence of the copies.**

To protect convergence of copies, participants can exploit an authenticated log of modifications. New participants have to retrieve the entire log in order to contribute. Unfortunately, the cost of joining a collaboration increases with the size of this log. Causal Stability allows to prune authenticated logs in a static collaborative group without any malicious participants.

In this paper, we first tailor Causal Stability to dynamic groups in presence of malicious participants. We then propose a mechanism to verify the consistency of a pruned log and a mechanism to authenticate a snapshot from a pruned log.

***Index Terms*—Authenticated log; Prunable log; Authenticated snapshot; Distributed collaborative systems; Dynamic group; Consistency; Convergence; Causal Stability; View-Fork-Join-Causal**

I. Introduction

Modern collaborative systems enable participants to co-author shared documents. They rely on optimistic replication algorithms [1] which support concurrent and conflicting edits of these documents. Participants hold their own copy of documents. They edit it locally before sharing their modifications with others. Thus, we witness successions of divergence and convergence between copies [2]. Convergence of copies confirms the propagation of the participants' modifications. It guarantees the progression of the collaboration. Hence, convergence is a liveness property for collaborative systems. If the system is unable to ensure convergence of copies, the collaboration is prone to failure. Sometimes, participants to a collaborative endeavor can find an advantage to make it fail. They could achieve that by making sure that two groups of participants maintain divergent views on a document. We need to ensure that it is not possible at a cost that is sustainable for the different stakeholders.

A solution used in operation-based replication is to authenticate the log of operations. Participants compare their respective logs to figure out which contributions they miss. Thus, honest participants will eventually get the same set of contributions and convergent logs. Combined with a concurrency and conflict resolver [1, 3, 4], copies of honest participants will eventually converge. In collaborative systems, new participants can join at any time. In order to obtain the current document state and contribute on it, they have to retrieve an authenticated log from a participant and verify its authenticity. Thus, the cost of joining a collaboration linearly grows with the size of the log. Furthermore, all participants have to store the log to be able to invite new participants in the group. Performance degrades as the size of the log increases. Reducing the footprint of authenticated logs in dynamic collaborative groups remains an important problem. One way to achieve that is to prune the log to maintain the cost at an acceptable level. Authenticated logs rely on consistency models [5] which define how contributions can be related to each other. Causal Stability [6] enables us to safely prune causal-consistent logs in static groups. Once a contribution is stable, participants have the guarantee that all participants included it in their log. They cannot issue a new contribution concurrently to a stable one.

In this paper, we propose to adopt the same strategy but with dynamic groups and in presence of an active adversary. To this end, we first adapt Causal Stability to dynamic groups. Then, we propose a consistency model that enables stabilization. Our consistency model is specified under the perspective of a log and relies on View-Fork-Join-Causal (VFJC) consistency model [7]. In the literature, VFJC is considered the strongest achievable consistency model in a distributed collaborative system with an active adversary. We define when a contribution is stable in our Stabilizable VFJC log. We design a mechanism to check the consistency of a pruned log. To illustrate the value of our stabilizable and prunable log, we propose a mechanism to authenticate a document snapshot from a pruned log.

II. System and adversary models

In this section, we introduce distributed collaborative systems along with basic notations. Then, we model the adversary of the system. Next, we present the definition of VFJC under the perspective of an authenticated log. Last, we illustrate the use of a VFJC log by the means of an example.

A. Distributed collaborative systems

Distributed collaborative systems support collaborative activities without a central server. A collaborative activity involves a group of participants that contribute to a common work. A

group has a finite and evolving number of participants. They can leave and join the collaboration at any time. We denote P the set of participants. $P = \{P_1, \ldots, P_i, \ldots, P_N\}$. Each participant owns an asymmetric signing key pair [8].

P is partitioned when there is no network connection between subsets of participants. A participant in a singleton is said to be offline. Subsets evolve depending on the status of their connection. The system is *always-available* [7] such that every participant can contribute to the common work at all time. To this end, each participant holds a copy of shared data. This is a *sticky availability model* [9]. S_t^i denotes the state of the copy held by a participant P_i at a discrete time t.

Participants apply contributions such as insertions and deletions on their own copy. c_j^i denotes the j-th contribution of P_i. c^i denotes any contribution of P_i. Participants eventually share their contributions to the rest of the group via an asynchronous network [10] without any coordination.

Always-availability leads to concurrent contributions and then conflicts [11]. As previous works [12, 7], we delegate the handling of concurrencies and conflicts to the application [1, 3, 4]. Causality tracking [13] exposes concurrencies between contributions. Instead of tracking every causal relation, we can only expose immediate dependencies [14, 15, 16]. Definition 1 introduces related notations. We assume that the contributions are causally delivered. Although the application decides how to resolve conflicts, we assume *strong convergence* [17, 3]: Two participants who play the same set of contributions get an identical state without any further coordination.

Definition 1 (Causality)**.** We write $a \rightarrow b$, if and only if a (causally) precedes b. We also said that b depends on a. "$\rightarrow$" is a strict partial order. Two contributions a and b are concurrent, denoted $a \| b$, if and only if neither precedes the other. b immediately depends on a, denoted $a \downarrow b$, if and only if there does not exist a contribution c such that a precedes c and c precedes b.

$$a \downarrow b \iff a \rightarrow b \wedge (\nexists c \cdot a \rightarrow c \rightarrow b) \qquad \text{(Immediate Dep.)}$$

$$a \| b \iff a \neq b \wedge a \not\rightarrow b \wedge b \not\rightarrow a \qquad \text{(Concurrency)}$$

B. Adversary model

We consider a *Byzantine adversary* [18]. The adversary controls an unbounded number of malicious participants. They can collude together. The adversary has also an extended control over network. She is able to alter, discard, duplicate, or reorder messages. She can partition the set of honest participants. However we assume the following limitations:

- She has limited resources, both in time and computational power. These resources are not sufficient to break the cryptographic primitives used, such as collision-resistant hash functions and digital signatures.
- She cannot prevent an honest participant from authenticating the public key of another honest participant.
- She is not able to prevent honest participants from eventually exchanging an unbounded number of messages.

The adversary aims at permanently prevent honest participants from eventually converging to the same state of shared data.

C. VFJC log as threat mitigation

Authenticated logs [19, 20] make misbehaviors evident. An authenticated log contains all participants' contributions along with their dependencies. Each contribution embeds the identifiers of its immediate dependencies. To make the log tamper-evident, the identifier of a contribution includes its collision-resistant hash and a contribution is digitally signed with the private key of its author.

H_t^i denotes the append-only log of a participant P_i at a discrete time t. For all t, H_t^i is a sublog of H_{t+1}^i. Figure 1 presents a representation of an authenticated log. If there exists a directed edge from a first contribution to a second one, then the second contribution immediately depends on the first one.

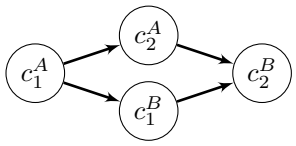


Figure 1. Authenticated log of Alice (P_A). Alice generates a first contribution c_1^A at discrete time $t = 1$. $H_1^A = \{c_1^A\}$. Then, she generates c_2^A such that c_2^A depends on c_1^A. $H_2^A = \{c_1^A \downarrow c_2^A\}$. Alice receives and delivers c_1^B, the first contribution of Bob (P_B). c_1^B is concurrent to c_2^A and depends on c_1^A. $H_3^A = \{c_1^A \downarrow c_2^A, c_1^A \downarrow c_1^B\}$. Then, she receives and delivers c_2^B, the second contribution of Bob. $H_4^A = \{c_1^A \downarrow c_2^A, c_1^A \downarrow c_1^B, c_2^A \downarrow c_2^B, c_1^B \downarrow c_2^B\}$.

Given an authenticated log, any honest participant can figure out who misbehaves from a set of rules. At the log level, the rules are summarized with a consistency model [5]. It restricts how contributions can be related. In the literature, VFJC is the strongest achievable consistency model in an always-available and convergent system in presence of a *Byzantine adversary*.

In VFJC, a participant is considered malicious only if she issues two or more contributions that are concurrent with themselves. To distinguish them from allowed concurrent contributions, i.e. those authored by distinct authors, we refer to them as non-linear[1]. Definition 2 defines when a participant is provably malicious.

Definition 2 (Provably malicious)**.** If a VFJC-consistent log has at least two non-linear contributions, then their author is provably malicious. Given a contribution c, $provablyMalicious(c)$ gives all provably malicious participants in the sublog that only includes c and all contributions that precede c.

$$provablyMalicious(c) = \left\{P_k \mid \exists (c_l^k \| c_m^k) \rightarrow c\right\}$$

By issuing non-linear contributions, a malicious participant creates concurrent branches of contributions called forks [17, 21]. The acceptance of forks in a log may raise difficult conflicts to resolve. VFJC limits the number of accepted forks according to the number of malicious participants.

Definition 3 presents VFJC under the abstraction of a log. Each honest participant maintains a VFJC log. This definition is different from the original one in the following aspects:

[1]Mahajan et al. [7] refers to them as non-serial

- VFJC was specified for distributed storage systems where several data are concurrently modified. In our system model, participants collaboratively edit a single data.
- VFJC consistency model uses real-time dates to prevent a past contribution from depending on a future one. We assume that the causal delivery layer enforces this.

Definition 3 (VFJC log)**.** Let a VFJC log $H_t^i = (Contrib_t^i, \rightarrow)$ from an honest participant P_i at a discrete time t. H_t^i is a locally finite partially ordered set (poset). $Contrib_t^i$ is the set of accepted contributions for P_i at time t. '$\rightarrow$' is a strict partial order with the following extra properties:

L1 (per-participant linear ordering) P_i, the owner of H_t^i, linearly orders her own contributions.

$$\forall c_k^i \ \forall c_l^i \cdot c_k^i \rightarrow c_l^i \iff k < l$$

L2 (proper generation — causality tracking) Every contribution of P_i, the owner of H_t^i, depends on all contributions of the log in which it was appended.

$$\forall c^i \exists t' \cdot H_{t'+1}^i - H_{t'}^i = \{c^i\} \wedge \forall x \in H_{t'}^i \cdot x \rightarrow c^i$$

L3 (sharing with no provably malicious) If there exists an immediate dependency relation between two contributions, then none of these contributions is authored by a provably malicious participant in the sublog that only contains them and all contributions that precede them.

$$\forall c^k \downarrow c^l \cdot P_k, P_l \notin provablyMalicious(c^l)$$

L4 (appendable — sharing with no provably malicious) There does not exist a pair of non-linear contributions such that one is among the latest contributions.

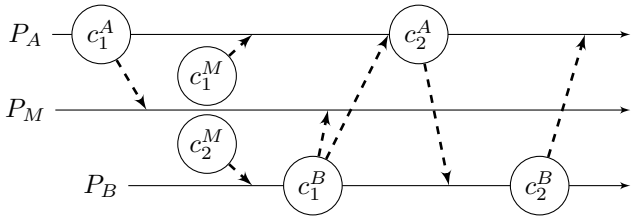
$$\forall c_l^k \cdot \nexists c_m^k \| c_l^k \wedge c_m^k \in latest_t^i$$
$$\textbf{where } latest_t^i = \{x \mid \nexists y \cdot x \rightarrow y\}$$

D. Motivating example

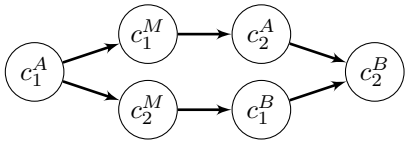
Suppose a team of volunteers, including Alice and Mallory, who organize an event. Alice creates a shared document and invites few team members to contribute to it. Each year, opponents try to disturb the event. Since the organizing team is open, malicious people, such as Mallory, can join the team. In order to protect convergence against misbehaviors, the document is secured thanks to a VFJC log.

Alice, Mallory, and other members participate in several editing sessions. The document contains minutes, reports, and a poll to know who is coming on the 4th of May, the date of the event. The log grows and all participants get a convergent log. Alice invites Bob in the editing group.

Figure 2 represents the part of the log of the collaboration when Bob is invited and can join the group. Alice (P_A) invites Bob (P_B) and ask collaborators to fill the poll. c_1^A sums up these actions. Bob joins the group thanks to Mallory (P_M). Mallory attempts to mislead Bob about the date. She modifies the date to the 8th of March and only shares this modification c_2^M with Bob. In the meantime, she confirms with c_1^M her availability on the 4th of May to Alice. Bob has the wrong date and then confirms with c_1^B his attendance on the 8th of March. Upon reception of c_1^B, Alice is aware that Bob has not the same causal context as hers. Given c_2^M, she figures out the misbehavior of Mallory. At the end of the depicted session, honest participants, Alice and Bob, get a convergent log and stop communicating with Mallory. Note that they stop communicating with Mallory as soon as they figure out her misbehavior: upon delivery of c_1^B for Alice, and c_2^A for Bob.



(a) Dashed arrows are transmissions of one or more contributions.



(b) A VFJC log with a fork junction at c_2^B.

Figure 2. (a) A collaborative session and (b) the VFJC log of honest participants (Alice P_A and Bob P_B) after their convergence. Mallory (P_M) issues two non-linear contributions ($c_1^M \| c_2^M$). She is provably malicious.

In their respective VFJC log, honest participants linearly order their own contributions (property L1) and expose the dependencies of their contributions (property L2). The linear order matches the order of generation, and the dependencies of a contribution are all contributions observed before the time of its generation. Since they send their contributions along with their dependencies, these orders are preserved. Hence, in any VFJC log, the contributions of every honest participant are linearly ordered and expose their dependencies. In the example, c_1^A precedes c_2^A ($c_1^A \rightarrow c_2^A$) and c_1^B precedes c_2^B ($c_1^B \rightarrow c_2^B$). c_1^B depends on c_1^A and c_2^M; while c_2^B depends on all contributions. Although c_1^B is received before the generation of c_2^A, c_2^A does not depend on it. According to the log, c_2^A depends only on c_1^A and c_1^M. We explain this behavior in following paragraphs.

Mallory authors two non-linear contributions $c_1^M \| c_2^M$ and thus breaks with property L1. From the point of view of an honest participant, another participant is not known to be malicious until misbehaviors are observed. Indeed, Alice and Bob do not know the misbehavior of Mallory when they receive respectively c_1^M and c_2^M. Since they have no proof that the received contribution is a non-linear one, they optimistically append it to their log. Alice and Bob hold two distinct forks.

Upon reception of c_1^B, Alice figures out the misbehavior of Mallory. Since Bob is not known to be malicious, Alice unavoidably accepts the fork $c_2^M \downarrow c_1^B$. However, if Alice appends this fork to her current log, she risks compromising her availability. Indeed, under L3, Alice is not able to directly join two forks, i.e. her next contribution cannot both depend on non-linear contributions (c_1^M and c_2^M) and immediately depends on one of them (c_1^M). However, in accordance with L2,

a contribution must properly expose its dependencies and then must depend on c_1^A, c_1^B, and the two non-linear contributions c_1^M and c_2^M. This contradiction leads to an impossibility to author new contributions.

L4 avoids this availability issue. Following L4, it is not possible to have two non-linear contributions such that one of them is at the end of the log. Since it is not conceivable to demand that Alice contribute each time that the situation is met, VFJC introduces special contributions called *views*. Before the integration of $c_2^M \downarrow c_1^B$, a *view* c_2^A is appended.

Upon reception of c_2^A, Bob accepts the fork $c_1^M \downarrow c_2^A$. L4 is fulfilled and then he does not need to issue a *view*. Bob's next contribution joins the two forks. All contributions that depend on a fork junction, including the junction itself, act as a proof of knowledge. Given such a contribution, we deduce that its author agrees on the fact that Mallory misbehaved. No more contributions of Mallory can be honestly accepted. Mallory is thus evicted of the group.

III. Log stability

In this section, we explore the concept of log stabilization. The log is divided into a stable part and an unstable part. The stable part includes only stable contributions. A contribution is stable once no more contributions concurrent to the included contributions can be appended to the log.

This section proposes to seek stability in dynamic groups and in presence of the adversary. We first assume the absence of an active adversary and tailor Causal Stability to dynamic groups. Then, we show that VFJC is not stabilizable in presence of malicious participants. Based on VFJC, we define a stabilizable log and View-Fork-Join-Causal Stability (VFJCS).

As stable contributions remain stable forever, they can be safely dropped from the log, we will make use of VFJCS to define a log pruning mechanism in the next section.

A. Provable Causal Stability in dynamic group

Causal stability [6] was defined in the context of static group and in absence of an adversary. Here, we tailor this concept to dynamic groups and we assume that all participants are honest.

A contribution is causally stable once all participants of the group have provably observed it. A participant provably observes a contribution if and only if she issues at least one contribution that depends on it. A convenient way to infer causal stability of a contribution is to show that no more concurrent contributions can be accepted. In dynamic group, participants join and leave the group at any time. A participant has an uncertain knowledge of the group composition. When a contribution is generated, it is therefore not possible to know all participants who are able to issue concurrent contributions.

In order to join and contribute to a collaboration, a newcomer has first to retrieve the log from another participant. Following L2, the newcomer inherits of all observations of that participant. If the participant has already observed a given contribution, then the newcomer cannot issue concurrent contributions. Conversely, if the participant has not already observed a contribution, then the newcomer may issue a concurrent contribution.

To determine who sent her log to a newcomer, we add a tracking mechanism based on invitations. To invite a newcomer, a participant issues an *invitation*. An *invitation* is a special type of contribution. $invitedIn(c)$ gives all participants invited in c. An *invitation* embeds public keys of the invited participants. For convenience, we introduce $previouslyInvited_H$:

$$previouslyInvited_H(c) = \bigcup \{invitedIn(x) \mid x \to c\}$$

A newcomer cannot contribute if he was not invited. To enforce this rule, we require that the author of a contribution had been invited in a preceding contribution. This rule does not apply for the initiator of the collaboration. Indeed, she issues a first contribution in which she is self-invited. It follows that every contribution, except the first one, has at least one immediate dependency. The log is bounded-below and its bottom element $\perp$ is the first contribution.

$$\forall c^k = \perp \cdot P_k \in invitedIn_H(c^k) \quad \text{(self-invited)}$$
$$\forall c^k \neq \perp \cdot P_k \in previouslyInvited_H(c^k) \quad \text{(previously invited)}$$

If the *invitation* of a new participant depends on a given contribution, then this participant can only issue contributions that depend on it. Conversely, if an *invitation* of a new participant precedes or is concurrent to a given contribution, then the participant may issue concurrent contributions. Participants who may issue concurrent contribution to a given contribution are required observers of the contribution. If all required observers of a contribution observe it, then the contribution is causally stable.

However required observers of a contribution can leave the collaboration. They may never observe the contribution. The contribution would never be stable. We need to take departures into account. A participant can leave the collaboration by requesting to be evicted. In order to evict someone, a participant issues an *eviction*. An *eviction* is a special type of contribution. $evictedIn(c)$ gives all participants evicted in c. Definition 4 names the participant evicted in a contribution c and in preceding evictions, the known evicted of c.

Definition 4 (Known evicted)**.** Let a contribution c from a VFJC log H_t^i. A participant is a known evicted in c if and only if she is provably malicious or she is evicted in the sublog that contains only c and predecessors of c.

$$evicted_H(c) = \bigcup \{evictedIn(x) \mid x \to c \vee x = c\} \cup provablyMalicious(c)$$

The acceptance of contributions whom the author was evicted follows consistency rules adapted from L3 and L4:

$$\forall c^k \downarrow c^l \cdot P_k, P_l \notin evicted_H(c^l) \quad \text{(sharing with no evicted)}$$
$$\forall c^k, c^j \in latest_t^i \cdot P_k \notin evicted_H(c^j) \quad \text{(appendable)}$$

An evicted participant cannot issue a contribution that depends on contributions that know her eviction. Thus known evicted of c are not required observers of c.

Required observers of a contribution can also be evicted in concurrent or following contributions. For the sake of simplicity, we consider only known evicted participants in following contributions. Only participants who have not yet provably observed a contribution e in which participants are known as evicted can directly accept contributions from these evicted participants. If all required observers of e have provably observed it, then it is no longer possible to accept contributions of known evicted participants of e. e is causally stable. All known evicted participants in a causally stable contribution are no longer required observers. Definition 5 summarizes which participants are required observer of a contribution.

Definition 5 (Required observer)**.** Let a contribution c from a VFJC log H_t^i. A required observer of c is either a participant invited in an invitation that precedes c or a participant invited in a concurrent invitation of c. All known evicted participants in c and in provably causally stable contributions that follow c are not required observers.

$$\begin{aligned} &requiredObs_H(c) = \bigcup \{invitedIn(x) \mid x \| c\} \\ &\quad \cup previouslyInvited_H(c) - evicted_H(c) \\ &\quad - \{evicted_H(x) \mid c \to x \wedge isProvablyCausallyStable(x)\} \end{aligned}$$

Under L1, honest participants linearly order their contributions. Thus if a participant issues a contribution which depends on another one, then she cannot issue a new contribution concurrent to these contributions. Thus a contribution is causally stable as soon as there exists proofs that its required observers have observed it.

The original definition of Causal Stability takes the implicit observation of the owner of the log into account. Thus, it is possible to have two identical logs in which the set of causally stable contributions is not identical. Since new participants retrieve and verify the log from other participants, it is important to define the stability of a contribution regardless of the initial owner of the log. We discuss more about this in section IV. Definition 6 defines Provable Causal Stability (PCS) which tailors Causal Stability to dynamic groups and removes implicit observations. Figure 3 shows how invitations interact with the provable causal stability status of a contribution.

Definition 6 (Provable Causal Stability)**.** In a VFJC log H_t^i, a contribution is provably causally stable if and only if all required observers have provably observed it.

$$\begin{aligned} &isProvablyCausallyStable_H(c) = \\ &\quad requiredObs_H(c) \subseteq provablyObs(c) \\ &\textbf{where } provablyObs(c^j) = \{P_j\} \cup \left\{P_k \mid \exists\ c^k \cdot c^j \to c^k\right\} \end{aligned}$$

B. Stabilizable-View-Fork-Join-Causal log

A malicious participant can issue a new contribution that is non-linear with one of her past contributions. If all participants provably observed past contributions, then following L3 and L4, they cannot honestly accept this new contribution.

However, any participant can invite someone. A malicious participant can issue an *invitation* which is non-linear with her past delivered contributions. If the invited participant contributes to the collaboration, then he issues a contribution that depends on this *invitation*. He is not provably malicious. Following L3 and L4, honest participants have to accept his contribution and by extension the fork. Figure 4 illustrates this scenario.

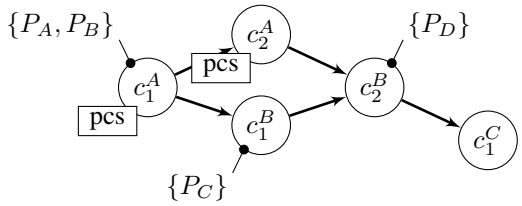


Figure 3. Upon generation of c_2^A, Alice knows that her (P_A) and Bob (P_B) are in the group. Thus, Bob and Alice are required observers of c_2^A. Upon delivery of c_1^B, Alice knows that Bob invited Carol. Since c_1^B is concurrent to c_2^A, Carol (P_C) could issue concurrent contributions to c_2^A. Hence, Carol is also a required observer of c_2^A. Upon delivery of c_2^B, Alice has a proof of observation of c_2^A from Bob and she is aware of the invitation of Dave (P_D). Since, c_2^B depends on c_2^A, Dave is not a required observer of c_2^A. Delivery of c_1^C gives a proof of observation of c_2^A from Carol. All required observers have observed c_2^A. Thus c_2^A becomes provably causally stable (pcs).

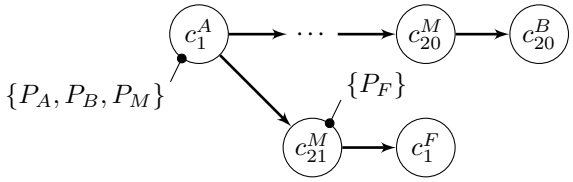


Figure 4. Alice (P_A), Bob (P_B), and Mallory (P_M) honestly issue contributions. Then Mallory issues an *invitation* c_{21}^M that invites Franck (P_F) and is non-linear with her past delivered contributions $c_1^M, \ldots, c_{20}^M$. Franck issues c_1^F. c_1^F depends on c_{21}^M. Since Franck is not provably malicious, honest participants accept the fork $c_{21}^M \downarrow c_1^F$. The log is VFJC-consistent.

The participation of newcomers in a fork leads to its acceptance by honest participants. Thus, it is always possible to have a concurrent contribution to another one. Log stabilization implies that at some point, contributions become stable, i.e. it is no longer possible to accept contributions that are concurrent to them. Hence, a VFJC log is not stabilizable in a dynamic group and in presence of malicious participants.

To stabilize a VFJC log, we have to reject forks that include only contributions of provably malicious participants and those of participants invited in the fork. This includes transitively invited participants. To simplify the expression of this property, we disable concurrent *invitations* of an identical participant. This can be obtained by including the collision-resistant hash of the *invitation* in the generated identifier of the participant. e.g. in Figure 3, the identifier of Carol consists in her public key and the hash of c_1^B. The owner of the public key can still be concurrently invited. He is attached to several participants.

A convenient way to reject these forks is to accept only contributions whom the author is known as participant both in concurrent branches of the contribution and in its immediate dependencies. In Figure 4, Franck is not known to participate in

c_{20}^B. c_{20}^B is the latest contribution of a branch that is concurrent to c_1^F. Thus c_1^F is not acceptable. To accept a contribution whom the author is unknown in concurrent branches, honest participants have first to accept the contribution that invites him. In our example, honest participants have to accept c_2^M. Following L3 and L4, they cannot accept the non-linear contribution c_2^M. Thus, they cannot accept the fork $c_2^M \downarrow c_1^F$.

Definition 7 (Known participants)**.** Let a contribution c from a VFJC log H_t^i. $participants(c)$ gives all invited participants in the sublog that includes only c and all contributions that precede c. Evicted participants in this sublog are not included.

$$participants_H(c) = previouslyInvited_H(c) \cup invitedIn(c) - evicted_H(c)$$

Definition 8 defines Stabilizable-View-Fork-Join-Causal (SVFJC) log. SVFJC strengthens VFJC by rejecting forks that preclude the stabilization of the log. Similar to property L4, property L8 avoids an availability issue.

Definition 8 (SVFJC log)**.** Let a SVFJC log $H_t^i = (Contrib_t^i, \rightarrow, \perp)$ from an honest participant P_i at discrete time t. H_t^i is a bounded-below VFJC log with $\perp$ its bottom contribution and with extra properties:

L5 (evicting only existing participants)

$$\forall c \cdot evictedIn(c) \subset \bigcup \{participants_H(x) \mid x \downarrow c\}$$

L6 (sharing with no evicted) If there exists an immediate dependency relation between two contributions, then neither is authored by an evicted participant in the sublog that contains only them and all contributions that precede them (Implies L3).

$$\forall c^k \downarrow c^l \cdot P_k, P_l \notin evicted_H(c^l)$$

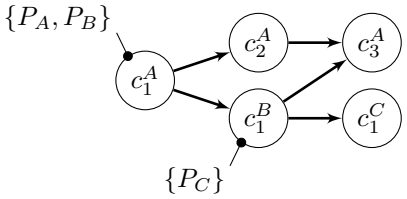
L7 (sharing with known participants) The author of c has to be known as participant in immediate dependencies of c. The author of an immediate dependency of c has to be known as participant in every immediate dependency of c.

$$\forall c^k \downarrow c^l \ \forall c^j \downarrow c^l \cdot P_l, P_k \in participants_H(c^j)$$

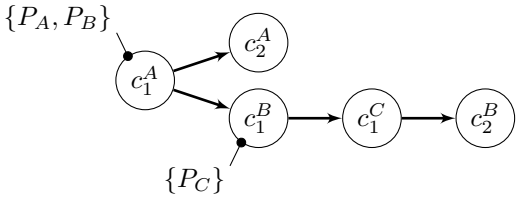
L8 (appendable — sharing with known participants) The author of every latest contribution has to be known as participant in all latest contributions. The owner of the log has to be a known participant of all latest contributions. This also excludes contributions whom the author is evicted.

$$\forall c^k, c^j \in latest_t^i \cdot P_i, P_k \in participants_H(c^j)$$
$$\textbf{where } latest_t^i = \{x \mid \nexists y \cdot x \rightarrow y\}$$

To illustrate the addition of SVFJC, we consider Figure 5 as an example. Under L8, an honest participant cannot accept a contribution in which the author is not known as participant in latest contributions of her log or a contribution which does not include in its known participants the authors of latest contributions. c_2^A does not include Carol in its known participants. Under L8, it is not possible to maintain a log such that both are latest contributions.



(a) Alice's SVFJC log.



(b) Bob's SVFJC log.

Figure 5. Bob (P_B) invites Carol (P_C) in his contribution c_1^B. Carol issues c_1^C which depends on c_1^B. In the meantime Alice (P_A) issues c_2^A which is concurrent with Bob's and Carol's contributions. Alice receives c_1^B and c_1^C, when Bob receives the contribution of Carol before the contribution of Alice. (a) and (b) respectively depicts the SVFJC log of Alice and Bob.

Alice and Bob have to issue a new contribution in order to respectively accept c_1^C and c_2^A. Alice issues c_3^A and then accepts c_1^C. c_3^A includes Carol in its known participants because it depends on c_1^B. Bob issues c_2^B and then accepts c_2^A. Like in subsection II-C, c_3^A and c_2^B may be *views*. Note that Carol cannot accept c_2^A without accepting a contribution which depends on c_2^A and c_1^B. c_3^A fulfills these requirements.

To enable stabilization, SVFJC logs reject forks in which new participants have contributed. This limits the availability of the system for these new participants. The application can use additional consistency rules to mitigate this issue. For instance, an invitation could be consistent only if it is accepted by a subset of trusted participants (author of the document, moderators, ...). This is out of scope of this paper.

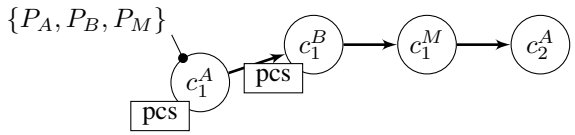
C. View-Fork-Join-Causal Stability

In presence of malicious participants, Provable Causal Stability is insufficient to stabilize a SVFJC log. Figure 6 shows a scenario in which a contribution is provably causally stable, when in the meantime, another honest participant accepts a contribution that is concurrent to this contribution.

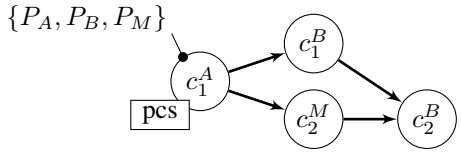
Only participants that have not provably observed a contribution c can directly accept a non-linear contribution of c. Hence, it is not possible to accept a non-linear contribution of c when c is provably causally stable. We say that the author of c is a provably linear observer of the sublog that contains only c and all contributions that precede c. Definition 9 defines the provably linear observers of a given contribution.

Definition 9 (Provably linear observer)**.** Let a contribution c of a SVFJC log H_t^i. A provably linear observer of c is a participant who is the author of at least one provably causally stable contribution that depends on c.

$$provablyLinearObs_H(c) = \{P_k \mid \exists c^k = c \vee c \rightarrow c^k \cdot$$



(a) Alice's SVFJC log.



(b) Bob's SVFJC log.

Figure 6. Alice (P_A) starts a collaboration with Bob (P_B) and Mallory (P_M). Mallory issues two non-linear contributions c_1^M and c_2^M. (a) In the log of Alice, c_1^B is provably observed by Mallory and Alice. From the point of view of Alice, c_1^B is provably causally stable (pcs). (b) Because Bob has not yet observed c_1^M, he accepts c_2^M in his log. Thus there exists a concurrent contribution to c_1^B which was honestly accepted.

$$isProvablyCausallyStable_H(c^k)\}$$

In a SVFJC log, a contribution is stable once it is no longer possible to honestly accept contributions that are concurrent to it. Hence, a contribution is stable if and only if all required observers of the contribution are also provably linear observers. Definition 10 names a stable contribution, a view-fork-join-causally stable contribution. Figure 7 illustrates how the collaboration of Figure 6 may evolve.

Definition 10 (View-Fork-Join-Causal Stability)**.** In a SVFJC log H_t^i, a contribution is view-fork-join-causally stable if and only if all required observers are provably linear observers.

$$isVFJCStable_H(c) = \\ requiredObs_H(c) \subseteq provablyLinearObs_H(c)$$

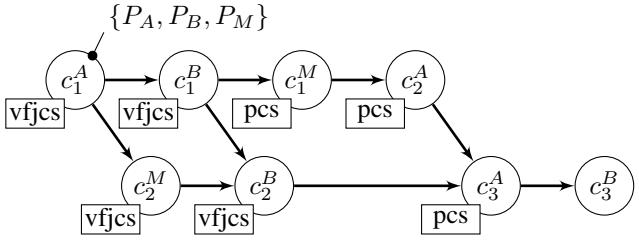


Figure 7. Mallory (P_M) is provably malicious. She is known as evicted in c_3^A. Since c_3^A is provably causally stable (pcs), then Mallory is no longer a required observer of any contributions. c_2^B precedes c_2^A and are pcs. Thus Alice (P_A) and Bob (P_B) are provably linear observers of c_2^B and all preceding contributions. These contributions have only Alice and Bob as required observers. They are thus view-fork-join-causally stable (vfjcs).

IV. Pruned log

In this section we describe how to prune the log and to verify its consistency. We first augment contributions in order to reduce the scope in which honest participants perform consistency verifications. Then we analyze which contributions can be dropped without hurting consistency verifications.

A strategy to prune the log is to remove as much as possible contributions. Only the contributions that are necessary to maintain the future consistency of the log must be kept. If an honest participant receives or generates a contribution, she must be capable of verifying its consistency with the log.

Before appending a contribution to her log, an honest participant verifies that properties L4, L5 (implies L3), L6, L7, and L8 hold. This verification requires the entire log or a sublog that includes the contribution and its predecessors. To remove contributions of the log, we have to reduce the size of the sublog in which its properties are verifiable.

These properties rely directly or indirectly on the functions $evicted_H$ and $previouslyInvited_H$. We propose to embed their result in every contribution. In practice, authenticated logs enable these data to be efficiently stored. $evicted_h$ and $previouslyInvited_h$ are the embedded versions of these functions. These changes enable the verification of all properties and the verification of the stability of a contribution in a sublog that contains only the appended contribution, its immediate dependencies, and all branches that are concurrent to at least one immediate dependency.

As seen in subsection III-C, a participant cannot honestly accept a concurrent contribution to a stable (view-fork-join-causally stable) contribution. It follows that we can remove every stable contribution, except when an unstable contribution may depend on it. For example, in Figure 7, c_1^A and c_2^M precede stable contributions. They are thus removable.

Definition 11 (SVFJC pruned log)**.** A log h_t^i is a SVFJC pruned log if and only if:

P1 (stable base) Every contribution of the base of the pruned log is stable.

$$\forall\ c^k \in base(h) \cdot isVFJCStable_h(c^k)\ \textbf{where} \\ base(h) = \left\{x \in h \mid \exists y \in deps(x) \wedge y \notin h_t^j\right\}$$

P2 (well-formed unstable contributions) Every unstable contribution is well-formed.

$$\forall \neg\ isVFJCStable_h(c^k) \cdot isWellFormed(c^k)$$

P3 (per-participant linear ordering) Similar to L1

$$\forall c_k^i\ \forall c_l^i \cdot c_k^i \rightarrow c_l^i \iff k < l$$

P4 (proper generation) Adaptation of L2

$$\forall \neg\ isVFJCStable_h(c^i)\ \exists t' \cdot \\ h_{t'+1}^i - h_{t'}^i = \left\{c^i\right\} \wedge \forall x \in latest_{t'}^i \cdot x \downarrow c^i$$

P5 (appendable — sharing with no provably malicious) Similar to L4

$$\forall c_l^k \cdot\ \nexists c_m^k \| c_l^k \wedge c_m^k \in latest_t^i$$

P6 (appendable — sharing with known participants) Adaptation of L8

$$\forall c^k, c^j \in latest_t^i \cdot P_i, P_k \in participants_h(c^j)$$

Definition 12 enumerates all consistency verifications that are locally performed on a contribution. Based on this definition, Definition 11 defines the consistency of a pruned log. $deps(c)$ provides the identifiers of the immediate dependencies of c.

Definition 12 (Well-formed contribution)**.** Let a SVFJC pruned log h_t^i from a SVFJC log H_t^i with $\perp$ the bottom contribution. A well-formed contribution c^k (we write $isWellformed(c^k)$) respects the following properties:

W1 (bottom identity) The contribution is either the bottom element or an intermediate contribution. An intermediate contribution has at least one dependency.

$$c^k = \perp \vee deps(c^k) \neq \emptyset$$

W2 (no transitive dependencies) The contribution exposes only its immediate dependencies.

$$\forall x, y \in deps(c^k) \cdot x \| y$$

W3 (known evicted) Previously evicted participants, new evicted participants, and new provably malicious participants are the only known evicted participants of the contribution.

$$\begin{aligned} evicted_h(c^k) = & \bigcup \{ evicted_h(x) \mid x \in deps(c^k) \} \\ & \cup evictedIn(c^k) \\ & \cup provablyMalicious(c^k) \end{aligned}$$

W4 (sharing with no evicted participants) Authors of the contribution and of its immediate dependencies are not known as evicted in the contribution.

$$\forall c^j \in deps(c^k) \cdot P_j, P_k \notin evicted_h(c^k)$$

W5 (Previously invited) Previously invited participants in c^k correspond to previously invited participants and newly invited participants in all immediate dependencies of the contribution.

$$\begin{aligned} & previouslyInvited_h(c^k) = \\ & \bigcup \{ previouslyInvited_h(x) \cup invitedIn(x) \mid x \in deps(c^k) \} \end{aligned}$$

W6 (sharing with known participants) The author of c^k is known as participant in all immediate dependencies. The author of every immediate dependency of c^k is known as participant in all immediate dependencies of c^k.

$$\forall c^j, c^l \in deps(c^k) \cdot P_l, P_k \in participants_h(c^j)$$

W7 (evicting only existing participants)

$$evictedIn(c^k) \subset \bigcup \{ participants_h(x) \mid x \in deps(c^k) \}$$

An honest participant is able to prune her log without hurting its future consistency or its availability. She can receive a new contribution and verify its consistency. She can also generate new consistent contributions. Because we define stability such that it is independent of the owner of the log, any participant can verify the consistency of a pruned log. Thus new participants can retrieve a pruned log and verify its consistency. Because contributions are missing, they cannot verify whether all contributions are well-formed. They have to trust verifications carried out by other participants. Every stable contributions are considered well-formed. If a new participant retrieves a pruned log, he is not able to obtain the current state of the document. Next, we propose to retrieve a pruned log with an untrusted snapshot and to authenticate the snapshot from the pruned log.

V. Authenticated snapshot

In this section we propose to authenticate an untrusted snapshot of the document with a SVFJC pruned log. To do that we augment *invitations* with state fingerprints.

When a collaboration starts, the log contains a single contribution. This first contribution $\perp$ is an *invitation* that invites the first participants. Each time a contribution is appended to the log, the contribution is played on the local copy of the shared document. Some contributions, such as *views*, *invitations*, and *evictions*, make no change to the state.

We say that a state S_t^i and a log H_t^i are consistent if and only if the causal execution of every contribution of H_t^i on the empty state S_0 gives S_t^i. Note that there exists several sequences of causal execution when the log contains concurrent contributions. In subsection II-A we assume *strong convergence* [17, 3]. This guarantees that every sequence of causal execution produces the same state. $undo(Contrib, S)$ causally undoes a set of contributions $Contrib$ on a state S.

Under L8, the latest contributions of a SVFJC log must include the owner of the log as known participants. An honest newcomer accepts only SVFJC-consistent pruned logs. Thus he must be a known participant in every latest contribution of the log. As long as a participant has not contributed to the collaboration, the contribution that invites him is unstable. Consistent pruned log includes all unstable contributions. It follows that the contribution that invites this newcomer is part of a consistent pruned log.

We are certain that an honest newcomer will observe the contribution that invites him. We can embed information to authenticate a snapshot in this contribution. When a participant invites a newcomer, she computes the fingerprint of the current state of the document and embeds it in the invitation. This fingerprint is a collision-resistant hash. $fingerprint(c)$ gives the embedded fingerprint if c is an invitation.

To verify whether the embedded fingerprint of an invitation c is correct, a participant has to produce the state observed by c. The participant first duplicates the current local state and undoes every concurrent and following contributions of c. Then she computes the fingerprint of the obtained state. If the computed fingerprint matches the embedded one, then the embedded fingerprint is correct. $computedFingerprint(S)$ is the function that computes the fingerprint of a state S.

To ensure that only correct fingerprints are embedded in invitations, in Definition 13 we extend the definition of well-formed contributions.

Definition 13 (Well-formed contribution and fingerprints)**.** Let a SVFJC log H_t^i and its attached state S_t^i. Let a SVFJC pruned log h_t^i from H_t^i. A well-formed contribution $c^k \in h_t^i$ fits:

W8 (correct fingerprint) If the contribution is an invitation, then the embedded fingerprint matches the computed fingerprint on the state observed by the contribution.

$$invitedIn(c^k) \neq \emptyset \implies$$
$$fingerprint(c^k) = computedFingerprint(S)$$
$$\textbf{where } S = undo(\{x \in h_t^i \mid x \| c^k \vee c^k \rightarrow x\}, S_t^i)$$

New participants retrieve a pruned log and a snapshot from a participant. By verifying the consistency of the pruned log, they also verify the consistency between the log and the snapshot. Thus, if the pruned log is SVFJC-consistent, then the snapshot is considered authentic.

A malicious participant can forge a state and embed the fingerprint of this state in one of her *invitations*. If the pruned log does not contain an *invitation* issued by an honest participant, then the new participants are not able to detect that the snapshot is not authentic. As seen in subsection III-B, the application can mitigate this using additional consistency rules. Another possibility is to embed the state fingerprint in more contributions. This is out of scope of this paper.

VI. Related work

Distributed storage systems store and atomically update several objects. The knowledge of the latest update of every object is sufficient to get the actual state. Based on this observation, *ASTRO* [22] proposes to sum up the older updates in order to shrink the log. In addition to securely tracking causal dependencies, honest participants have to maintain a balanced *Merkle tree* [23] in which the leaves are their own updates. Instead of directly signing an update, honest participants sign the root of their own *Merkle tree*. Every signature is a summary of one or more updates of a participant. *ASTRO* respects a weaker consistency model than VFJC. It does not limit the number of accepted forks. A malicious participant can conceal non-linear contributions in a summary. It is not possible for an honest participant to determine if a summary is well-formed. Our consistency model limits the number of accepted forks.

Based on the same observation, *Depot* [19] proposes a checkpoint mechanism to discard old updates. Periodically, a client suggests update removals. An honest client needs the approval of every client in order to garbage collect the suggested updates. A malicious client can prevent the garbage collection since a consensus is required. In contrast, we propose to prune the log without any coordination and extra communications; honest participants incrementally prune their log. Like in *Depot*, a malicious participant can prevent the garbage collection. However, in practice, we can evict inactive participants while *Depot* cannot determine if a disagreement to a consensus is honest or dishonest. To perform consistency checks, *Depot* includes enough information in the checkpoints. We design augmented contributions in order to perform consistency checks in a pruned log. Our proposition could be adapted to *Depot*.

Instead of sending the entire log, *SPORC* [24] propose the transmission of a snapshot. Periodically, a participant publishes a signed snapshot on a central server. Participants trust each other, thus the snapshot is considered authentic if the signature is correct. Under our adversary model, this approach is not secured: a malicious participant can freely tamper the snapshot.

Zhao et al. [25] introduces an optimistic Byzantine Fault-Tolerant (BFT) approach for collaborative editing. Unlike common agreement-based BFT systems, their approach first applies the execution step and runs the agreement step on-demand. Indeed, honest server replicas execute and broadcast updates to client replicas as soon as they are received. When an honest client does not receive enough matching updates, it demands an agreement among the pool of server replicas. The paper describes a mechanism to join a collaboration using a snapshot of the document. To add a new client, a server replica runs an agreement over the clients. Once accepted, honest clients send an identical snapshot. A majority of matching snapshot is enough to consider a snapshot authentic. If the number of active clients is not enough, the application gives up safety in profit of liveness. In contrast, we propose an invitation mechanism that enables us to asynchronously accept newcomers. Our joining process requires a single snapshot. We rather abort a joining process than sacrifice safety.

VII. Discussions

A. Causal stability

The usefulness of our proposal relies on the assumption that the size of the stable part of the log is close to the entire log.

Our adversary model assumes that two honest participants can eventually exchange messages and does not limit how long eventually is. Thereby, messages could not be delivered before the end of the collaboration. Participants could be disconnected for a long period time. As long as they are disconnected, the stabilization of the log is not possible.

Although the selected model is worthy for the design of secure mechanisms, it is not relevant for their assessments. Indeed, a system where participants have poor interactions is likely not a collaboration. In practice, inactive participants can be evicted from the group after a given period. With such a mechanism, stabilization can be enforced. Only contributions shared between inactive participants could be lost.

B. Availability vs Consistency

Distributed systems make a trade-off between consistency and availability. Availability is a usability requirement and then cannot be traded for strong consistency guarantees [7, 19]. Weak consistency guarantees are not suitable for every application [26]. As mitigation, we build on VFJC [7], the strongest achievable consistency guarantees in an always-available and convergent system.

VIII. Conclusion and future work

Distributed collaborative systems rely on replication algorithms to support concurrent modifications of shared documents in presence of network partitions. Convergence of document copies determines the liveness of a collaboration and can be protected with authenticated logs against an active adversary.

The cost of joining a collaboration linearly grows with the size of the authenticated log.

We have shown that it is possible to protect convergence of document copies with a pruned version of the authenticated log. We designed a mechanism based on the authentication of snapshots of the document using a prunable authenticated log of the contributions.

To this end, we extended Causal Stability to dynamic groups (in which members can join and leave for their convenience) and in presence of an active adversary. Our proposal relies on an adaptation of the View-Fork-Join-Causal (VFJC) consistency model to a stabilizable version of the log of contributions. We also showed that the log can be pruned without threatening its future consistency. To illustrate the value of our stabilizable and prunable log, we proposed to authenticate a snapshot from a pruned log. The maintenance of a pruned log and its transmission along with a snapshot reduce the footprint of authenticated logs when new participants join the collaboration.

To enable stabilization, our authenticated log rejects forks in which new participants have contributed. This limits the availability of the system for these new participants. A future work could explore extensions to offer more guarantees to these participants. For instance, we could add time-based or trust-based rules to accept an invitation under conditions.

Our system model elevates partitioning as a feature: partitions can independently collaborate. A future work could explore log stabilization in partitions.

Acknowledgements

We would like to thank Matthieu Nicolas, Olivier Perrin, Alizée Dulliand, Rado Randrianomanana, Alexandre Merlin, and anonymous reviewers for their meaningful comments.

References

[1] Y. Saito and M. Shapiro, "Optimistic Replication", *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005. DOI: 10.1145/1057977.1057980.

[2] P. Dourish, "The Parting of the Ways: Divergence, Data Management and Collaborative Work", in *Proceedings of the 4th European Conference on Computer-Supported Cooperative Work (ECSCW 1995)*, Kluwer Academic Publishers, 1995, pp. 215–230. DOI: 10.1007/978-94-011-0349-7_14.

[3] M. Shapiro, N. Preguiça, C. Baquero, *et al.*, "Conflict-Free Replicated Data Types", in *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of distributed Systems (SSS 2011)*, 6976, vol. 6976, Springer, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[4] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems", in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989, pp. 399–407. DOI: 10.1145/67544.66963.

[5] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems", *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, 19:1–19:34, 2016, ISSN: 0360-0300. DOI: 10.1145/2926965.

[6] C. Baquero, P. S. Almeida, and A. Shoker, "Making Operation-Based CRDTs Operation-Based", in *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques*, Springer, 2014, pp. 126–140. DOI: 10.1007/978-3-662-43352-2_11.

[7] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, Availability, and Convergence", Department of Computer Science, The University of Texas at Austin, Tech. Rep. TR-11-22, 2011.

[8] A. Salomaa, *Public-Key Cryptography*. Springer, 1996. DOI: 10.1007/978-3-662-03269-5.

[9] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Trade-offs in Replicated Systems", *IEEE Data Engineering Bulletin*, vol. 39, no. 1, pp. 14–26, 2016.

[10] N. A. Lynch, "Distributed Algorithms", in. The Morgan Kaufmann Series in Data Management Systems, 1996, ch. 14, pp. 457–471, ISBN: 1558603484.

[11] T. Mens, "A State-of-the-Art Survey on Software Merging", *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, 2002. DOI: 10.1109/TSE.2002.1000449.

[12] C. Weilbach, K. Kühne, and A. Bieniusa, "Decoupling Conflicts for Configurable Resolution in an Open Replication System", *ArXiv:1508.05545 [cs]*, 2015.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. DOI: 10.1145/359545.359563.

[14] R. Prakash, M. Raynal, and M. Singhal, "An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments", *Journal of Parallel and Distributed Computing*, vol. 41, no. 2, pp. 190–204, 1997. DOI: 10.1006/jpdc.1996.1300.

[15] S. E. Pomares Hernández, J. Fanchon, and K. Drira, "The Immediate Dependency Relation: An Optimal Way to Ensure Causal Group Communication", in *Annual Review of Scalable Computing, Editions World Scientific, Series on Scalable Computing*, vol. 6, 2003, pp. 60–79.

[16] S. E. Pomares Hernández, "The Minimal Dependency Relation for Causal Event Ordering in Distributed Computing", *Applied Mathematics & Information Sciences*, vol. 9, no. 1, pp.57–61, 2015. DOI: 10.12785/amis/090108.

[17] P. Viotti and M. Vukolić, "Consistency in Non-Transactional Distributed Storage Systems", *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, 19:1–19:34, 2016. DOI: 10.1145/2926965.

[18] L. Lamport, R. E. Shostak, and M. C. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982. DOI: 10.1145/357172.357176.

[19] P. Mahajan, S. Setty, S. Lee, *et al.*, "Depot: Cloud Storage with Minimal Trust", *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, 12:1–12:38, 2011. DOI: 10.1145/2063509.2063512.

[20] H. T. T. Truong, C.-L. Ignat, and P. Molli, "Authenticating Operation-based History in Collaborative Systems", in *Proceedings of the 17th ACM International Conference on Supporting Group Work (GROUP 2012)*, ACM, 2012, pp. 131–140. DOI: 10.1145/2389176.2389197.

[21] D. Mazières and D. Shasha, "Building Secure File Systems out of Byzantine Storage", in *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, ACM, 2002, pp. 108–117. DOI: 10.1145/571825.571840.

[22] P. Mahajan, S. Lee, A. Zheng, *et al.*, "ASTRO: Autonomous and Trustworthy Data Sharing", Department of Computer Science, The University of Texas at Austin, Tech. Rep. TR-09-29, 2008.

[23] R. C. Merkle, "Secrecy, Authentication, and Public Key Systems", AAI8001972, PhD thesis, Stanford University, Stanford, CA, USA, 1979.

[24] A. J. Feldman, W. P. Zeller, M. J. Freedman, *et al.*, "SPORC: Group Collaboration Using Untrusted Cloud Resources", in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, USENIX Association, 2010, pp. 337–350, ISBN: 978-1-931971-79-9.

[25] W. Zhao, M. Babi, W. Yang, *et al.*, "Byzantine Fault Tolerance for Collaborative Editing with Commutative Operations", in *2016 IEEE International Conference on Electro Information Technology (EIT)*, IEEE Computer Society, 2016, pp. 0246–0251. DOI: 10.1109/EIT.2016.7535248.

[26] D. Terry, "Replicated Data Consistency Explained Through Baseball", *Communications of the ACM*, vol. 56, no. 12, pp. 82–89, 2013. DOI: 10.1145/2500500.