Practical Client-side Replication: Weak Consistency Semantics for Insecure Settings

Albert van der Linde
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa
a.linde@campus.fct.unl.pt

João Leitão
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa
jc.leitao@fct.unl.pt

Nuno Preguiça
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa
nuno.preguica@fct.unl.pt

ABSTRACT

Client-side replication and direct client-to-client synchronization can be used to create highly available, low-latency interactive applications. Causal consistency, the strongest available consistency model under network partitions, is an attractive consistency model for these applications.

This paper focuses on how client misbehaviour impacts causal consistency. We analyze the possible attacks to causal consistency and derive secure consistency models that preclude different types of misbehaviour. We propose a set of techniques for implementing such secure consistency models, which exhibit different trade-offs between the application guarantees, and the latency and communication overhead.

Our evaluation shows that secure consistency models impose low overhead when compared with their insecure counterparts, while providing low user-to-user latency and server load compared with traditional client-server architectures. Secure consistency models can be used to enrich server-based architectures with fast and secure peer-to-peer interactions.

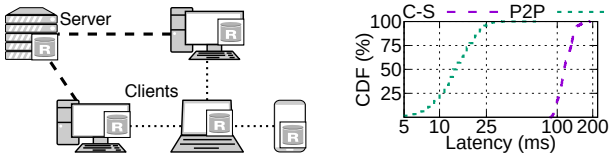
PVLDB Reference Format:
Albert van der Linde, João Leitão, and Nuno Preguiça. Practical Client-side Replication: Weak Consistency Semantics for Insecure Settings. *PVLDB*, 13(11): 2590-2605, 2020.
DOI: https://doi.org/10.14778/3407790.3407847

1. INTRODUCTION

Latency is a key property in distributed applications, with several studies showing that user engagement drops when latency increases [17, 18, 43]. In interactive applications, such as multi-user games, low latency is vital for a good user experience. Geo-replication [88, 53, 81] is used to reduce latency of global services, by keeping replicas at multiple geographic locations and allowing clients to access the closest replica.

In such cloud computing architectures, client interactions are mediated by servers. Communication latency among clients is directly related to the latency between clients and the server, which can be high due to the small number of data centers where applications can be deployed.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407847



(a) Peer-to-peer model. (b) P2P vs. C-S latency.

Figure 1: Peer-to-peer architecture (latency for Peer-to-Peer (P2P) and Client-Server (C-S) deployments, with a server on AWS Ireland and clients in Grid5000 in France – see §6).

An alternative is to allow clients to locally replicate application state and synchronize directly among themselves using peer-to-peer interactions (Figure 1a) – several recent works adopt this approach [42, 86, 83]. As shown in Figure 1b, direct client-to-client interactions potentially lead to lower latency when compared with the traditional approach where all communication is mediated by a server. Additionally, clients can continue interacting given than they can communicate directly, masking network and server faults.

Many applications can benefit from the direct interaction among nearby clients, from collaborative applications, such as document editors, games and audience engagement applications, to location-dependent information sharing, such as geo-social networks, traffic information, and contact tracing applications. A particularly interesting example is that of location-based and augmented reality games, such as Ingress and Pokemon Go [65, 66], where a player interacts with nearby players and low latency is crucial for interactivity.

Moving application state to clients and allowing peer-to-peer synchronization poses multiple security challenges. First, it is necessary to guarantee that unauthorized accesses do not compromise confidentiality and integrity. This problem has been addressed resorting to standard security techniques [58, 89]. Second, it is necessary to address client misbehaviours, which can be characterized by the Byzantine [47] and BAR [50] models. Several algorithms for providing functionality under these models have been proposed, such as reliable dissemination [15, 50, 74] and BFT state machine replication [16, 90, 10]. Decentralized replication algorithms, such as secure causal BFT [31, 30, 75] and blockchain-based replication [4, 14, 33, 82], still enforce a total order on all operations, imposing an high latency on writes.

This paper focuses on a different problem: how to address clients' misbehaviour, that deviates from correct behaviour in a way that cannot be detected, in systems with client-side replicas and peer-to-peer synchronization and that adopt

weaker consistency models to promote availability and low latency. This is an important issue as many applications (e.g., games) require high availability and low latency for a smooth user experience, and users have incentives for misbehaving (to gain an unfair advantage), but only if it is not possible to prove that they are misbehaving. To the best of our knowledge, this work is the first to study the interaction between weak consistency models and misbehaving replicas.

We analyze the possible effects of misbehaving nodes in causal consistency [3, 53], the strongest consistency model that is available under network partitions [57]. From this analysis, we derive secure variants of causal consistency that preclude different types of misbehaviour. We propose practical algorithms for implementing these models in a setting where clients communicate directly. We also propose a secure version of eventual linearizability [76, 85], as a way to provide stronger guarantees when required.

We have designed and evaluated a system that provides the proposed secure consistency models. Our evaluation shows that adopting the secure consistency models imposes low overhead when compared to an insecure version, while improving latency and server load when compared to classic client-server architectures. The latency gains are more expressive for interactions among nearby clients, which maps the expected use in many applications, such as augmented reality games. We show that providing multiple secure consistency models can be important, as it allows developers to select a different point in the trade-off space between application guarantees, latency, and communication overhead. In summary, this paper makes the following contributions:

- a systematic study on how client misbehaviour impacts the guarantees of causal consistency (§3);
- the definition of secure variants of causal consistency, preventing multiple types of misbehaviour (§4);
- algorithms for the secure variants of causal consistency and also eventual linearizability (§5); and
- an experimental evaluation (§6) of our prototype.

2. SYSTEM MODELS

In this paper we consider multi-user applications where users interact using their own devices. We assume that the state of the application is maintained by a centralized service and that clients replicate a subset of the state. Clients execute operations on their local replicas and synchronize directly among themselves by propagating operations (the terms client and replica are used interchangeably throughout this paper). A subset of these clients synchronize with the server to ensure durability and to allow clients that cannot communicate directly with other clients to participate.

To provide low latency and ensure availability despite network and replica faults, the system is designed to provide causal+ consistency [3, 53]. Eventual convergence is achieved by modeling state as operation-based CRDTs [78], and by executing operations in every replica in causal order.

Some of the proposed solutions assume that the system includes a set of low-resource trusted infrastructure nodes.

2.1 Attacker Model

In this paper we study how malicious clients can thwart the guarantees of causal consistency. We consider an attacker model focused on clients (i.e., servers and other infrastructure nodes are trusted). Correct clients will always follow the prescribed protocol. Malicious clients can behave

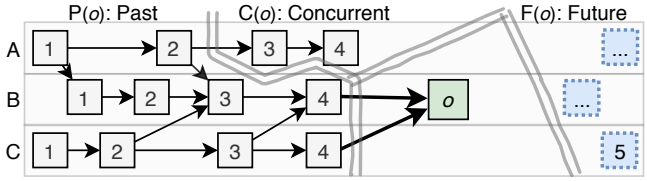


Figure 2: Dependencies in causal consistency.

in a fully byzantine mode (arbitrarily deviating from their prescribed behaviour) or be rational [50], meaning they will deviate from the prescribed protocol (to attempt to gain some advantage) only if the misbehaviour cannot be detected by correct clients or servers. We assume that the cryptographic primitives such as digital signatures and hash functions cannot be undermined. This is similar to the BAR model [50] enriched with a trusted centralized server and infrastructure nodes with limited resources.

A byzantine node [47], by not focusing in hiding its misbehaviour, provides to its peers either a demonstration or a proof of its malicious action. For example, sending unsigned messages is a demonstration of misbehaviour but doesn't lead to a proof, while a correctly signed message that contains a falsehood is a proof-of-misbehaviour. Although we do not focus on byzantine behaviour, when it occurs, including attempts to interfere with operation propagation, the system detects and isolates such clients, ensuring correct clients continue to communicate (§5.7). We assume that malicious clients cannot prevent correct clients from establishing secure channels with a server or among themselves.

The primary focus of our study is the risks posed by rational clients. These clients may attempt to manipulate the generation and propagation of operations in a way that benefits them. This problem is important, for example, for the game industry, where peer-to-peer approaches are attractive in terms of latency and availability if cheating can be avoided. In this case, clients (players) have interest in being rational (to gain an advantage) if they cannot be discovered as being rational (to avoid being banned due to cheating). In §3 we detail the possible attacks by misbehaving clients.

2.2 Consistency model

Causal consistency is a consistency model that can be described, at a high level, as enforcing clients to always observe a state that respects happens-before relationships among operations [46]. For a replicated system, we say that operation o_1 happened before operation o_2, $o_1 \prec o_2$, iff o_2 was generated in some replica where o_1 had already been executed. For a set of operations Ops, $(Ops, \prec)$ is partial order.

For any operation o, generated at replica r, we can consider three disjoint sets of operations, as shown in Figure 2:

- $P(o)$ is the set of past operations that happened before o – this is also known as the causal history of o, $H(o)$;
- $C(o)$ is the set of operations that are concurrent with o, i.e., $\forall o_c \in C(o), \neg(o_c \prec o) \wedge \neg(o \prec o_c)$;
- $F(o)$ is the set of future operations that happened after o, i.e., $\forall o_f \in F(o), o \prec o_f$.

We say that for a set of operations Ops, $O_i = (Ops, <)$ is a causal serialization of $O = (Ops, \prec)$ iff O_i is a linear extension of O, i.e., $\forall o_1, o_2 \in Ops, o_1 \prec o_2 \Rightarrow o_1 < o_2$. A system enforces causal consistency iff all replicas execute operations according to a causal serialization.

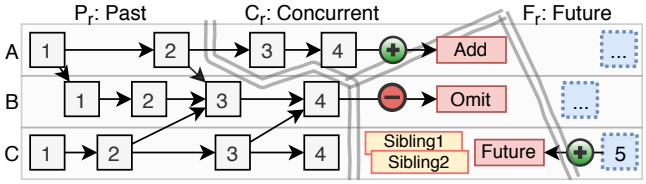


Figure 3: Attacks on causal consistency.

Multiple algorithms have been proposed to enforce causal consistency (and causal dissemination) [46, 7, 12, 32, 53, 79]. Two of the most popular techniques consist in using version vectors [60, 67] and direct dependency graphs [53, 69, 72].

In the former, the dependencies of each operation are summarized in a vector that states which operations generated at each site happened before a given operation. In the example of Figure 2, the dependencies of the new operation o would be [2,4,4], stating the dependency on operations up to 2 from replica A, and up to 4 from both replicas B and C. Using direct dependencies, each operation includes information on the operations that have been executed before its generation. Since each operation includes its dependencies, it is possible to build the whole dependency graph. In the example, operation o would depend on {B:4, C:4}.

3. ATTACKS ON CAUSAL CONSISTENCY

Malicious replicas, if left unchecked, can easily disrupt the properties of a replication algorithm for causal consistency. This section systematically identifies possible attacks.

3.1 Tampering with other replicas' operations

This class of attacks comprises actions that a malicious replica can perform regarding operations created by other replicas – this includes tampering with the integrity of messages in transit (such as modifying causal dependencies of operations), generating operations in the name of other replicas, and creating malformed operations.

A simple example is altering the overall order of events by creating new operations and set other (already existing) operations to depend on them. In a game where players shoot each other, one can make a shot depend on a later created moving operation, making the shot miss instead of hit the malicious player. In general, such attacks can be addressed by having replicas sign the operations they generate, as discussed in §5.1. Attacks on message propagation (e.g., not propagating some operations) are discussed in §5.7.

3.2 Attacks on operation generation

This class of attacks consists in manipulating the creation of new operations by attaching incorrect dependencies. Figure 2 illustrates the correct dependencies of a new operation o, while Figure 3 shows possible attacks, discussed next.

3.2.1 Omitting dependencies

A malicious replica may create an operation that contains a subset of the actual dependencies. Given the set of operations executed in replica r, P_r, this attack consists in setting the causal history of a new operation to the set ${P_r}^{rem}$, such that ${P_r}^{rem} \subsetneq P_r$ (i.e., ${P_r}^{rem} \subset P_r \wedge \exists o \in P_r : o \notin {P_r}^{rem}$).

By including only a subset of the known operations in the dependencies, a malicious replica can forge an operation that is concurrent to operations that it already knows, creating an operation that occurred in its logical past. In Figure 3 this is shown as *Omit*, with the omission of known operations from replica B in the dependencies of the new operation.

This attack can also be used for moving away from another user's shot (as the previous attack). The difference is that the result will depend on the application's conflict resolution policy, as the shot and movement will be concurrent.

We note that the possible attacks to causal consistency is similar when using version vectors or direct dependencies. Consider the example of Figure 2. When generating o, operations 1-2 from replica A and 1-4 from replicas B and C were received, leading to a version vector [2,4,4] or to the direct dependencies $\{B:4; C:4\}$. When using version vectors, a malicious replica can selectively remove a suffix of operations from any replica – e.g., vector [0,4,4] would remove the dependencies from replica A. However this has no impact when enforcing causal consistency, as only the direct dependencies are important – executing operation 3 (and 4) from B requires executing operation 2 from A before.

3.2.2 Depending on unseen operations

A malicious replica may create an operation that depends on an operation that has not been executed locally and possibly does not even exist yet. Given the set of operations executed in replica r, P_r, this attack consists in setting the causal history of a new operation to the set ${P_r}^{add}$, such that $P_r \subsetneq {P_r}^{add}$ (i.e., $P_r \subset {P_r}^{add} \wedge \exists o \in {P_r}^{add} : o \notin P_r$). In Figure 3, *Add* and *Future* represent, respectively, depending on a not yet received operation and on a future operation. Such an attack allows a malicious replica to create operations that do not respect the real time order, potentially counter-acting the actions of other replicas even before they take place.

Consider a multi-player game where players fight some monster, which drops a specific item when defeated. The item is available as soon as it is dropped, but only the first player to react will pick it up. A malicious player can create a *pick-up* operation that depends on a future drop operation. This ensures an unfair advantage over other players, as the pick-up will execute immediately after the drop operation is created and the dependency is met.

Depending on the application, such an attack may be detectable – the dependency may never be executed (e.g., if the creature is never defeated) or might reach the replica that will execute the dependency before its execution.

3.2.3 Combining omit and add

A malicious replica can create an operation that combines the previous two attacks, altering dependencies to both omit and include unseen operations. Given the set of operations executed in replica r, P_r, the attack consists in setting the causal history of a new operation to the set ${P_r}^{a+r}$, such that $(\exists o_1 \in {P_r}^{a+r} : o_1 \notin P_r) \wedge (\exists o_2 \in P_r : o_2 \notin {P_r}^{a+r})$.

3.2.4 Sibling generation

In any replicated system, an operation typically includes an identifier. A malicious replica may generate two different operations with the same identifier (sibling operations). In a system that is not prepared to deal with malicious replicas, different replicas may execute different versions of the operation, leading to a permanent state divergence.

A simple example is a replica creating, within a chat application, two different messages with the same identifier, leading to different users seeing a different chat history.

4. SECURE CONSISTENCY MODELS

We now propose consistency models derived from causal consistency that deal with the presence of malicious replicas, by addressing the attacks discussed in the previous section.

4.1 A Secure Form of Causal Consistency

We start by deriving a secure form of causal consistency, defining a set of properties that must be enforced. Our first property precludes tampering with the causal history of an operation, after it is generated:

SECURE CAUSAL PROPERTY 1. *(Immutable History) If H_o is the causal history of operation o at generation, o is delivered with H_o at every correct replica.*

We now define properties concerning the dependencies on unseen operations. The *No Future Dependencies* property precludes having dependencies on operations that have not been executed yet in any replica (*Future* in Figure 3):

SECURE CAUSAL PROPERTY 2. *(No Future Dependencies) Given P_{all}, the set of all operations generated in any replica of the system when operation o_{new} is created, o_{new} can only depend on such operations, i.e., $\nexists o \notin P_{all} : o \prec o_{new}$.*

As for creating operations that depend on operations generated in some other replica that have not been executed locally (*Add* in Figure 3), we note that this situation is equivalent to synchronizing the local replica before issuing the operation. In a system where replicas synchronize peer-to-peer, verifying this situation is not straightforward. Thus, we only define a property that enforces all correct replicas to execute operations respecting their defined dependencies:

SECURE CAUSAL PROPERTY 3. *(Causal Execution)All correct replicas execute an operation respecting the dependencies defined in the operation. Given the causal history of an operation o, $H(o)$, the causal serialization $O_r = (Ops, <)$ in every correct replica r is such that $\forall o_i \in H(o), o_i < o$.*

We now address the problem of having a malicious replica issuing two operations with the same identifier, which can lead correct replicas to execute different versions of the operation (*Siblings* in Figure 3). This could be avoided by executing a consensus step to certify the operation associated with each identifier [30, 31, 75]. However, this goes against our objective of allowing a replica to immediately execute a received operation, thus being highly available. Instead, we require this situation to be eventually detected and reported to correct replicas, by locally executing a *fault*(o) operation:

SECURE CAUSAL PROPERTY 4. *(Eventual Sibling Detection) Given two operations o_1 and o_2 with the same identifier, for any replica r that has executed the set of operations Ops_r, the following conditions will apply: (i) if $o_1 \in Ops_r$, then eventually $fault(o_1) \in Ops_r$, with $o_1 < fault(o_1)$; and (ii) if $o_2 \in Ops_r$, then eventually $fault(o_2) \in Ops_r$, with $o_2 < fault(o_2)$.*

4.1.1 Omitting dependencies

We now consider the attack where a malicious replica generates an operation that omits some of the locally executed operations from the set of dependencies (*Omit* in Figure 3). It is impossible for a correct replica r_c, receiving an operation o from a malicious replica r_m, to verify if o includes all dependencies it should or not. Even if r_c has previously sent to r_m some operation o_o, the fact that o_o is not included in the dependencies of o can be due to o being generated before o_o was received and executed. The correct behaviour due to delays is indistinguishable from an incorrect behaviour where o_o is purposely omitted from the dependencies.

Let $O^{real} = (Ops, \prec^{real})$ be the happens before partial order as registered by an external observer, which perceives all dependencies within the system. When generating a new operation o, a malicious replica may omit some of its real dependencies, leading to the partial order $O^{omit} = (Ops, \prec^{omit})$, that omits some of the dependencies defined in O^{real}, including direct dependencies of o and indirect dependencies among other operations established through o.

A malicious replica cannot omit dependencies in an arbitrary way without being detected. This can be exemplified in Figure 3, where if operation A2 is omitted from the dependencies, then all operations that depend on it (A3, A4, B3, B4) must also be omitted. When using version vectors, as discussed before, setting the dependencies as [0,4,4] has no actual effect, as B3 has A2 as its dependency and so will only execute after A2 has executed. Moreover, it allows the detection of the misbehaviour by analyzing the graph of dependencies. When using direct dependencies, it is also only possible to omit (a suffix of) immediate dependencies.

In general, to avoid detection, a malicious replica cannot omit from the dependencies of an operation o it generates any operation o_p that happened before an operation o_m included in the dependencies of o:

SECURE CAUSAL PROPERTY 5. *(Limited Omission) Given P_r, the set of operations executed in replica r, a malicious replica can only omit dependencies for a new operation without being detectable, by setting the causal history of the new operation to be ${P_r}^{rem}$, such that ${P_r}^{rem} \subsetneq P_r \wedge \nexists o_p \in P_r \setminus {P_r}^{rem}, o \in {P_r}^{rem} : o_p \prec o$.*

We now formally define *secure causal consistency*:

DEFINITION 4.1. ***Secure Causal Consistency*** *is a model which ensures that any correct replica r executes operations according to a serialization order $O_r = (Ops, <)$, such that:*

- *a) no operation with tampered dependencies is executed (*Immutable History*);*
- *b) no operation that depends on a future operation is executed (*No Future Dependencies*);*
- *c) O_r is a valid serialization of $(Ops, \prec^{omit})$, i.e., given the set of dependencies of operation o, $H(o)$, $\forall o_p \in H(o), o_p < o$ (Causal Execution, Limited Omission);*
- *d) if two different operations with the same identifier are generated, any correct replica that executes any of such operations o will also eventually execute $fault(o)$, with $o < fault(o)$ (Eventual Sibling Detection).*

4.2 Strengthening Secure Causal Consistency

Ideally, a replica should be forced to set the real dependencies to the operations it generates:

SECURE CAUSAL PROPERTY 6. *(Real Dependencies) The dependencies of an operation o, $H(o)$, are the* real dependencies *iff $\forall o_p, o_p \prec^{real} o \Rightarrow o_p \in H(o)$.*

This leads all operations to be created with the dependencies according to $\prec^{real}$ (we discuss practical implementations of such a property in §5.3, leveraging trusted software or hardware). From this property we can derive:

DEFINITION 4.2. ***Secure Strict Causal Consistency*** *is a consistency model that ensures that any correct replica i executes operations according to a serialization order $O_i = (Ops, <)$, such that O_i is a valid serialization of $(Ops, \prec^{real})$, i.e., given the dependency set of operation o, $H(o)$, $\forall o_i \in H(o), o_i < o$ (Causal Execution). Note that a), b) and d) from Secure Causal Consistency are enforced by this model.*

4.3 A Compromise for Collusion Tolerance

Even if replicas are unable to generate operations with incorrect dependencies, two colluding replicas can communicate through a side-channel, circumventing the mechanisms to enforce secure strict causal consistency.

A possible solution to tackle this challenge is to use a consistency model based on recency, requiring new operations to depend on all existing operations at all replicas (e.g., External Causal Consistency [11]). Implementing this approach requires some form of synchronization among all replicas for generating an operation. This goes against our goal of remaining available in the presence of network partitions [36].

We adopt a different approach: operations are generated and executed without coordination, and a replica is eventually notified if an operation o_2, that might externally depend on operation o_1, was executed before o_1.

We define a total order, $<^{ext}$, that guarantees that if an operation o_2 might depend on operation o_1, then $o_1 <^{ext} o_2$. The total order must respect the following properties:

EXTENDED CAUSAL PROPERTY 1. *(Total Order) Given the set of operations Ops, $O_{<^{ext}} = (Ops, <^{ext})$ is a total order (i.e., $\forall o_1, o_2 \in Ops : o_1 <^{ext} o_2 \vee o_2 <^{ext} o_1$).*

EXTENDED CAUSAL PROPERTY 2. *(External Causal Visibility) Given two operations o_1 and o_2, if some replica has observed (in realtime) o_1 before the generation of o_2 in any replica, then $o_1 <^{ext} o_2$ ($\forall r_1, r_2, \forall o_1, o_2 : observed_{r_1}(o_1) <^{obs} generate_{r_2}(o_2) \Rightarrow o_1 <^{ext} o_2$, with $<^{obs}$ the total order of events as observed by an external omniscient observer).*

We now define two consistency models that use this total order ($<^{ext}$). First, Secure Extended Causal Consistency which extends Secure Causal Consistency by notifying applications of out of order (according to $<^{ext}$) executions:

DEFINITION 4.3. ***Secure Extended Causal Consistency*** *is a consistency model that ensures that any correct replica i executes operations according to a serialization order $O_i = (Ops, <)$, such that:*

a-d) equal to Secure Causal Consistency;

e) if an operation executes in an order that violates $<^{ext}$, the application is notified when executing the operation that should have been executed earlier using signal, i.e., if $\exists_{o_2} : o_1 <^{ext} o_2 \wedge o_2 < o_1$ then the execution of o_1 is replaced by the execution of signal(o_1, O_a), with O_a the operations that should have been executed after o_1 according to $<^{ext}$ (i.e., $O_a = \{o : o < o_1 \wedge o_1 <^{ext} o\}$).

Note that signaled operations are not necessarily a causality violation due to collusion, and should be handled by the application in an application-dependent way. For example, in a chat application, an out of order reception can simply lead to an update in the user interface showing clearly what happened, leading to an intuitive user experience. In §5.5 we further detail how this model can be used in practice.

Second, Secure Eventual Linearizability[76] which enforces the execution of received operations according to $<^{ext}$:

DEFINITION 4.4. ***Secure Eventual Linearizability*** *is a consistency model that ensures that any correct replica i executes the operations it has received, Ops, according to the serialization order $O_i = (Ops, <^{ext})$.*

Note that this definition allows to execute operations immediately after they are received. To enforce the order $<^{ext}$, when a new operation is received, it might be necessary to undo/redo operations. In the next section we discuss how the proposed secure consistency models can be implemented.

5. ALGORITHMS FOR SECURE MODELS

This section discusses possible implementations of the secure consistency models, which we used in our prototype. Figure 4 presents excerpts of the proposed algorithms.

We assume that each replica authenticates with the centralized service when joining the system (i.e., starting its session), receiving a certificate signed by the server that can be used to prove the replica's identity. All replicas trust the server, being able to locally validate certificates of other replicas. Replicas use the associated private key to sign information. We further assume that replicas communicate through secure channels, authenticating each other by leveraging the certificates obtained when joining the system.

5.1 Authenticity, Non-Repudiation and Integrity

An application can issue a new operation, *op*, by calling function `NewOp` (line 5). This function creates an operation record that includes the operation with its identifier (pair (cnt_{rep}, id_{rep})) and metadata specific for each consistency model. The record has a signature of this information ($rec.sign$). The operation record (or simply operation where no confusion can arise) is sent to the replica's neighbours.

The signature is used to ensure that operations propagated among replicas are originated in a valid replica (authenticity), that operations can be associated to its creators (non-repudiation), and that they are not modified in transit by malicious replicas (integrity).

Upon receiving an operation record (line 21), a replica first verifies that the signature is correct. If the operation's signature cannot be validated, the operation is discarded. If the signature is valid, the metadata is verified and, if valid, processed according to the chosen secure consistency model.

If the previous verifications end successfully, the operation contents can still be invalid, for instance when a Byzantine node issues invalid operations according to application logic or impersonates other replicas [21, 27, 91]. In this case, a proof-of-misbehaviour is produced for the replica that generated the operation and sent to the centralized infrastructure which will disseminate the proof among all clients to expel the Byzantine client from the system (§5.7).

For misbehaviour that does not allow to produce a proof (e.g., sending unsigned messages), clients will simply disconnect from its sender. Continuous erroneous execution thus leads to a malicious replica to be restricted to pure client-server model as correct replicas will deny connections to it.

5.2 Secure Causal Consistency

To track causal dependencies, the metadata of each operation includes the identifiers of its direct causal dependencies (line 11) [53, 69, 72]. For a newly generated operation, the direct causal dependencies includes any locally executed operation o for which there is no operation $o_n : o \prec o_n$. When

```
1: Local State:
2:     id_rep : identifier of local replica
3:     cnt_rep : counter used for identifying operations
4:     cert_rep : local replica certificate
5: procedure NewOp(op)
6:     cnt_rep ← cnt_rep + 1
7:     rec.op ←AddDeps_model(op, (cnt_rep, id_rep))
8:     rec.sign ←sign_rep(rec.op)
9:     SEND(p, rec), ∀p ∈ neighbours
10: function AddDeps_Secure(op, op_id)
11:     deps ← LatestDepIds( )
12:     hDepRecs ← Hash(LatestDepRecord( ))
13:     return <op, op_id, random( ), deps, hDepRecs>
14: function AddDeps_StrictSecure(op, op_id)          ▷ Run in
15:     deps ← LatestDepIds( )                        ▷ secure module
16:     return Encode(<op, op_id, random( ), deps>)
17: function AddDeps_ExtSecure/EvtLinear(op, op_id)
18:     deps ← LatestDepIds( )
19:     (ts, sign_TiS) ← TiS_OpTs(<Hash(op, deps)>)
20:     return <op, ts, deps, sign_TiS>
21: upon receive rec from p do:
22:     if VERIFY(cert_p, rec) then
23:         if CheckDeps_model(p, rec) then
24:             Process_model(rec)
```

Figure 4: Algorithms for secure consistency models.

compared with version vectors, this approach has the advantage of not requiring one entry per replica, being more suitable for handling large and dynamic memberships.

Immutable History: As replicas sign the operations, the causal histories of operations cannot be manipulated by malicious replicas, thus enforcing Secure Causal Property 1.

No Future Dependencies: To disallow depending on operations that have not yet been seen, including future operations, the metadata of an operation includes a random number (line 13) and a cryptographic summary (hash) of all direct causal dependencies (line 12). This makes it impossible for a malicious replica to create a dependency on an operation that it has not yet observed, as it is unable to compute a valid hash of the dependencies. A replica validates the hash before executing the operation. If an invalid hash is detected, a proof-of-misbehaviour for the replica that generated the invalid operation is issued which, as previously stated, will lead the replica to be excluded from the system. This technique allows to enforce Secure Causal Property 2.

Causal Execution: Causal execution (Secure Causal Property 3) is achieved by verifying, before executing an operation, that its dependencies have already been executed.

Our prototype uses the protocol proposed by Linde et. al. [86], where operations are disseminated through a communication overlay and a replica only propagates an operation to a peer after propagating the operation's dependencies (or knowing that the peer already received them). This guarantees that, when receiving an operation, causal dependencies are satisfied and the operation can execute immediately.

When a replica detects that a remote replica is not following the protocol, it produces a proof-of-misbehaviour. This proof follows directly when discovering an out-of-order propagation as all messages sent between two replicas are hash-chained: the signed message includes the hash of the previous message (omitted in the code for simplicity).

Limited omissions: As the metadata includes only direct dependencies, it is impossible by design for a replica to introduce causal gaps in the dependency graph (as it can only omit dependencies in a suffix of the dependencies), thus guaranteeing Secure Causal Property 5.

Eventual Sibling Detection: We use several techniques to detect when multiple operations with the same identifier are created, as to enforce Secure Causal Property 4. First, a replica that receives two siblings from different paths creates a proof-of-misbehaviour and informs the server. Second, an operation includes in its metadata the hash of its dependencies – when receiving an operation, if this hash does not match the hash computed locally for the same dependencies, the replica signals a potential sibling by informing the server. Finally, the server periodically sends a summary of its state, containing the hash of the last observed operations at the server that replicas can use to verify if they have received the same operations. If the verification fails, the client connects to the server to verify the hashes of each individual operation, leading to a proof-of-misbehaviour. The server's state summaries also follow causal propagation – a replica found to propagate a summary which includes some operation it has not previously sent, is proven to be malicious.

While these mechanisms cannot prevent Byzantine replicas from exhibiting arbitrary behaviour, they are enough to prevent rational replicas (that want to avoid exclusion from the system) to perform such attacks. These mechanisms together allow to provide **Secure Causal Consistency**.

Intuitively, there is no defense against omitting operations or delaying operation propagation. As any two operations generated by the same replica always have an implicit dependency from higher to lower identifier, a replica is unable to selectively hold back its own operations. Nevertheless, replicas can collude to omit operations from causal dependencies. The following sections discuss how to provide additional guarantees by leveraging trusted components, being it within trusted servers or secure hardware modules.

5.3 Secure Strict Causal Consistency

Secure Strict Causal Consistency requires a replica to record the exact causal dependencies. This can be implemented by delegating the reception and generation of operations to a trusted service. An instance of the service can use trusted hardware, such as Intel's SGX [61], if available at the replica. When the replica has no trusted hardware, the same function can be delegated to instances of the trusted service running at nearby replicas or infrastructure nodes.

Intuitively, the service is responsible to receive operations before delivering them at the client to track all received operations (to guarantee that causal dependencies are faithfully assigned). When new operations are created, the service assigns the precise dependencies. When compared with *secure causal consistency*, the metadata does not need to include the random number and the hash of dependencies, as: (i) the trusted module will never include incorrect dependencies; and (ii) it detects if an application tries to use the same identifier for two different operations.

To prevent the application from accessing an operation before it being processed by the secure module, each operation is ciphered (line 16) by the secure module with a key shared only among instances of the trusted service. As only instances of the trusted service can access the shared key, it is guaranteed that operations were correctly created and can only be accessed after being Decoded by the secure module (for simplicity, the *Decode* step is omitted in the code).

An immediate issue that arises when using an external service is ensuring correct client-handover and maintaining causal dependencies when such a handover is done. Applying earlier works [1, 2, 72] is not practical as no thought has been given to nodes attempting to circumvent causal relations – intuitively, any reconnection by a client from one instance of the service to another always requires some form of coordination among instances of the service. A malicious client actively *hopping around* can introduce a great overhead in such a system. Due to this costly operation, we do not further explore this direction, leaving it for future work.

5.4 Collusion Tolerance

Even if a single replica is unable to forge an operation with incorrect dependencies, two replicas, r_1 and r_2, can collude create an operation o_2 that is concurrent with some known operation o_1 (by having replica r_1 sending o_1 through an external channel to r_2, and having r_2 submitting o_2). In §4.3, we proposed to address this problem by using a total order $<^{ext}$ on operations, such that if o_2 might depend on o_1, then $o_1 <^{ext} o_2$ – we synthesized the underlying properties as Extended Causal Property 1 and 2.

Consider realtime timestamps uniquely attributed to each operation by an omniscient entity. Such timestamps would allow to order operations in a way that respects the required conditions, as any externally visible happens-before relations are captured by the global realtime order. Having a single server assigning timestamps would provide the required total order, but would defeat the goal of allowing replicas to make progress without coordination.

5.4.1 Practical external visibility

We propose the use of a decentralized timestamping service (TiS) of which all instances have synchronized clocks with a small divergence of up to δ. Replicas communicate with this service to obtain timestamps for their operations.

There is a total order on all operations generated at a single instance of the service, as we restrict it to only emit one timestamp per time unit. With multiple instances, we order operations with colliding timestamps by sorting on a hash of the whole operation. This provides a total order among all operations, thus providing Extended Causal Property 1.

For enforcing Extended Causal Property 2, we must guarantee that, if any replica observes an operation, and after (in realtime), any replica creates a new operation, the latter must appear after the former in the total order.

Secure timestamps for $<^{ext}$: Consider Figure 5, where replica r_2 generates o_2 after observing replica r_1's o_1. If clocks and execution were perfect, the lowest timestamp for o_2 would be: $t_2 = t_1 + (TiS \rightarrow^T r_1) + (r_1 \rightarrow^T r_2) + (r_2 \rightarrow^T TiS)$[1]. As clocks are not perfectly synchronized, we need to consider the divergence of clocks, leading to: $t_2 = (t_1 + \delta_{TiS_1}) + (TiS \rightarrow^T r_1) + (r_1 \rightarrow^T r_2) + (r_2 \rightarrow^T TiS) + \delta_{TiS_2}$.

As in a distributed systems it is impossible to exactly measure the minimum value for $(TiS \rightarrow^T r_1)$,$(r_1 \rightarrow^T r_2)$, and $(r_2 \rightarrow^T TiS)$, the safe approach is to assume they are 0. This leads to $t_2 = t_1 + \delta_{TiS_1} + \delta_{TiS_2}$.

For guaranteeing that $t_2 > t_1$, we force all TiS instances to wait $TWait$ before returning a newly generated timestamp to a client (i.e., timestamp t_i is returned at $t_i + TWait$), with $TWait > |\delta_{TiS_1} + \delta_{TiS_2}|$. As δ_{TiS_1} is the clock divergence of TiS_1 to a global reference, $\delta_{TiS_1} + \delta_{TiS_2}$ is the clock divergence between TiS_1 and TiS_2. To guarantee that the condition for $<^{ext}$ holds for timestamps generated in any pair of TiS instances, we need to wait for more than the maximum clock divergence between any pair of TiS instances, i.e., $max(\{|\delta_{TiS_i} + \delta_{TiS_j}| + 1, \forall_{TiS_i,TiS_j} \in T\})$.

[1] $A \rightarrow^T B$ is the minimum time it takes for a message to be sent from A to B, by any means, internal or external to the system.

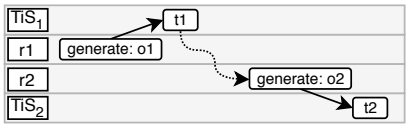


Figure 5: Timestamping.

5.4.2 Discussion on TiS implementation

The TiS is a lightweight service that executes across multiple (and potentially geographically distributed) nodes, allowing a replica to request a signed timestamp for a given operation. To avoid that a malicious replica requests multiple timestamps that are then used at its convenience to issue arbitrary operations at points in the past, these timestamps must be issued linked to a particular operation. To this end, when a replica requests a timestamp for an operation, it must send the TiS the cryptographic summary of the operation and dependencies (line 19). The TiS will then issue a verifiable and trusted timestamp in the form of a tuple $(ts, sign_{TiS})$, where $sign_{TiS}$ is the signed operation summary (including ts) and ts is the timestamp generated by the TiS. These timestamps can be validated by any replica using the certificate of the TiS, which is signed by the centralized server. In summary, the service can be implemented by leveraging well known timestamping protocols [5, 38].

The deployment of the timestamping service is an interesting research question on its own. The lightweight nature of the TiS, in contrast to full application servers, makes it easy to deploy and scale. We envision the following scenarios: *i*) collocated with the application's centralized service; *ii*) geo-distributed at multiple cloud points of presence; *iii*) at edge locations such as ISPs and 5G towers; or *iv*) on client devices within Trusted Execution Environments.

A client that cannot contact a TiS instance directly or through other replicas must stop generating operations (and possibly notify the user). Given the different alternatives to deploy the TiS, we expect this situation to be rare (and less frequent than unavailability in client-server architectures).

Synchronizing TiS instances: TiS instances should execute a clock synchronization protocol (e.g., NTP [62] or PTP [40]), whose precision will directly impact $TWait$. This work does not focus on synchronizing the clocks of TiS instances, and we assume that: (i) TiS clock synchronization cannot be tampered by clients [23]; (ii) under normal conditions and in most common deployments scenarios, $TWait$ will be under single-digit milliseconds [35, 44, 48, 63, 71]. We note that even if $TWait$ is up to double-digit milliseconds, it still allows faster progress than resorting to a client-server model.

5.5 Secure Extended Causal Consistency

Applications that use **Secure Extended Causal Consistency** will be notified when an operation is delivered in an order that is correct to the system's observed causality, but that does not respect the external observer's order, $<^{ext}$. The system itself does not reject or re-order any operation –

it is up to the application to use this information to perform suitable actions in accordance with the application logic. A simple example, discussed previously in §4.3, is a chat application where the user-interface can be updated with this information for an intuitive outcome.

As reported in other systems [68], using this information can lead to complex application code, as it typically requires applications to recompute the final state, while guaranteeing that all replicas converge to the same state. To help programmers, our prototype includes a set of CRDT datatypes that take advantage of this information by providing semantics that are not typically available in systems that provide causal consistency. Some examples include:

- a *list* object where concurrent insertions on the same position are ordered by insertion time – e.g., this can be useful in a chat application;
- a *map* with a first-to-write-wins conflict resolution policy – e.g., this can be useful for ordering bids of the same value in an auction.

If using this information is too complex, the application programmer should resort to Secure Eventual Linearizability, in which operations are executed according to the external observer's order (by relying on undo-redo).

5.6 Secure Eventual Linearizability

Secure eventual linearizability is implemented by executing the operations in timestamp order, thus guaranteeing that each replica executes the received operations according to $<^{ext}$, as defined by the timestamps obtained from the TiS. As operations can be received out-of-order, our implementation uses an undo-redo execution model – when an operation is received, operations that were executed out-of-order are undone and reapplied in the correct order. Although this consistency model only uses the timestamps, the metadata of an operation includes its dependencies (line 18) as this information is often useful for managing the application state.

With this approach, the final outcome of an operation is only determined after its execution order becomes stable, i.e., when no operation with a smaller timestamp can arrive – until then, the operation's execution is considered tentative [85]. Our prototype includes an optional mechanism for establishing the stability of operations, that works as follows. The server periodically defines the stability timestamp, t_s (up to some seconds in the past of the current clock), and determines the set of operations that become stable: the operations it has received with a timestamp t, such that $t \leq t_s$. This information is broadcasted to all clients (we note that it is only necessary to propagate the identifiers of operations that have no operation that happened after). The operations with a timestamp $t \leq t_s$ that are not included in the set of stable operations are undone forever. When a replica finds out that an operation it has generated is in this situation, it may resubmit the operation by first obtaining a new timestamp from the TiS.

5.7 Denial of Service and Eventual Delivery

For implementing the proposed secure consistency models, it is also necessary to guarantee that all correct replicas will receive all valid operations. We build on the mechanism to detect sibling operations to achieve this property. The centralized service periodically disseminates a summary of its state, including a hash of the last observed (concurrent) operations. If after some time, a replica has not received the reported operations or its own operations have not been reported by the centralized service (either as a recent operation or a dependency of a recent operation), the replica contacts the server directly to synchronize its state. The replica also contacts the server when it does not receive an updated server report for some time.

This approach prevents Eclipse Attacks, where malicious replicas attempt to create a barrier between correct replicas. When this happens, correct replicas will communicate resorting to replica-server-replica interaction. Several decentralized protocols for providing secure broadcast [29, 50, 59] and preventing Eclipse attacks [80] have been proposed in literature and could have been adopted to provide the same guarantees without resorting to the centralized service.

Since in our system the server must be used to connect to the peer-to-peer network (§5.7), it also acts as a protection against malicious replicas appearing with multiple identifiers (Sybil attacks [28, 64]) or from reappearing with a new identifier after a proof-of-misbehaviour has been generated.

A correct replica that detects a Byzantine behaviour immediately disconnects from the malicious one, never to connect again. If a malicious replica misbehaves with all correct peers, it is eventually removed from the peer-to-peer network and limited to interact through the server. If a proof can be obtained, then the correct replica submits the proof to the server which propagates it through the peer-to-peer network. This is similar to the eviction protocol in Legion [86], leading Byzantine replicas to have no effect on correct ones.

A malicious replica can follow the protocol but submit invalid operations according to the application logic. As an operation includes its dependencies, other replicas and the server can use this information to verify the validity of the operation. The complexity of this process depends on the application. When verification is complex (e.g., requiring recomputing the state of the replica when the operation was issued), this can be done in the background by the server.

This opens the opportunity for attacks, but there is little incentive for a single rational client to emit an invalid operation, as it will be expelled from the system and it will not be able to reenter. However, when users compete in teams, if the gain/loss ratio is high enough, there might be an incentive to sacrifice some members by issuing an invalid operation that will only be detected much later – simple solutions such as banning the whole team can be applied but that depends on the application. This is an open problem that is not specific to our decentralized approach, but inherent to systems running under weak consistency.

Our system mitigates attacks on the consistency model but it is not able to completely eliminate all of the attacks that have been discussed. Although eventual delivery is enforced, as we show in the evaluation (§6.6), it still requires a 50% ratio of correct replicas to ensure interactions among correct replicas are not delayed. This is in line with the work on BAR-tolerant protocols [50].

6. EXPERIMENTAL EVALUATION

Our evaluation demonstrates that security guarantees can be provided even when clients replicate data and communicate directly. The highlights of our results are the following:

- Our secure consistency models provide considerable reduction in user-to-user latency when compared to client-server, with interactions between nearby clients exhibiting the larger improvements. (§6.2)

- Our solution scales to a large number of clients with a modest increase in latency. (§6.3)
- Low-latency is crucial for effective Eventual Linearizability since its execution is affected by staleness. (§6.4)
- Deployment of the timestamping service should consider the geographic distribution of clients, ideally by exploiting dynamic placement in client vicinity. (§6.5)
- The proposed algorithms disallow discoverable Byzantine misbehaviour, mitigating the effect of such behaviour on correct clients. Rational clients (remaining undiscovered) can impact correct clients most when they are a majority of the nodes. Resorting to the TiS and the centralized component mitigates the actions of colluding rational replicas. (§6.6)

Prototype: Our prototype allows applications running in browsers to use the secure causal consistency models proposed, namely secure causal and secure extended causal consistency, and for applications that require stronger guarantees, secure eventual linearizability (*EvtLin*).

Our prototype extends Legion [86], a causally consistent system that includes a library of CRDTs [87] for merging concurrent updates. The secure models were implemented by replacing Legion's propagation mechanisms, which were susceptible to the attacks discussed in §3, and extending the CRDT library to implement the semantics defined in §5.5. Legion is used in the evaluation as the baseline providing unsecured causal consistency (*Causal unsecured*).

The prototype was written in JavaScript, with test application running in browsers or as NodeJS applications. The latter were primarily developed to avoid the overhead of graphical user interfaces in our experiments. Our prototype has an abstract communication layer that enforces the required security abstractions and FIFO channels on top of the network layers in both environments: WebSockets (TLS) in NodeJS and WebRTC (DTLS) in browsers. The implementation of our algorithms uses SHA-256 and RSA:2048 for hashes and signatures, relying on the forge library [25].

Baseline: User-facing applications typically rely on client-server architectures. To serve as a baseline, we added support in our prototype for strict-serializability, using client-server interactions with a single master (*Client-Server* in the plots). Switching from *EvtLin* to *Client-server* requires changing a single configuration variable, ensuring a fair comparison as there are no hidden language overheads or implementation optimizations that can affect results.

Replicas communicate with the server using standard techniques: WebSocket over TLS, with an initial client authentication (user login). Unlike the secure models, operations are transmitted over the secure channel without any further cryptographic processing. The server verifies if an operation is valid before propagating it to other replicas, by checking if it can be applied in the current state.

Application: For the evaluation, we created a game where players move on a 2D map with obstacles and gather coins to obtain points. Initially a coin is located at the center of the map, and players are spawned near the edges. When a player touches the coin, it gains a point and a new coin is spawned at a random location on the map (computed deterministically from the hash of the catching replica's operation). The movement and gathering operations can be verified based on the last movement from that player (this gives a verified trace as they start in deterministic positions).

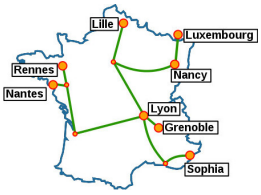


Figure 6: Grid5000 clusters map (France).

6.1 Experimental Setup

Our experiments were run on the Grid'5000 [6] platform (G5k), complemented with AWS EC2. Clients instances (1792 by default, with 256 at each cluster) are spread over G5k's clusters (except Grenoble), shown in Figure 6. Each client has one CPU core and 1 GB RAM available. Unless stated otherwise, the server runs in Ireland (AWS EC2 t2.xlarge, 4vcpu, 16GB) and the TiS instances are deployed in Ireland, Paris, and Frankfurt (also on AWS EC2).

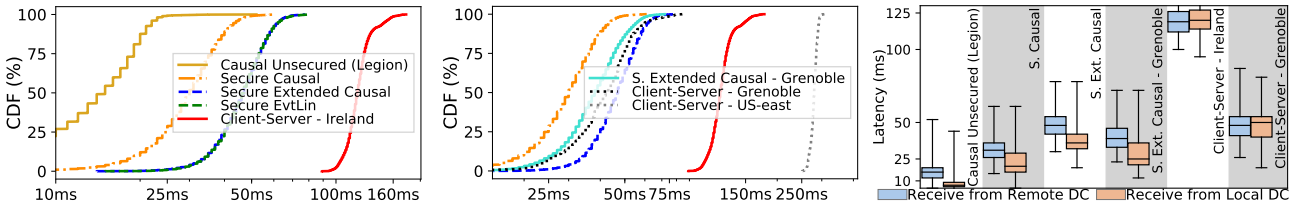
With this setup, clients and servers are distributed across different geographic locations, with different latency among clients and from different clients to the servers.

6.2 Latency Evaluation

Client-to-client latency measurements consider the time since the call to create a new operation (`NewOp`) until its reception at each client. All clients generate, apply locally, and propagate an operation every 5 seconds. Thus, every *client* will receive, verify, and apply $1792/5 = 358.4$ operations per second. A typical (single room) multiplayer game has significantly less players and operations being executed.

Figure 7a reports the latency (as a Cumulative Distribution Function) observed for the secure variants of our system compared with the client-server baseline. Among the secure variants, Secure Causal provides the lowest latency between clients as new operations only have to be signed before being propagated (and verified upon reception). Extended Causal and EvtLin require a client to obtain a timestamp from the TiS for each operation, leading to an additional delay before propagation. As a consequence, results are very similar (due to this reason, we omit the results of EvtLin in Figs. 7b, 7c, and 8). Client-Server presents the highest latency due to the time required for operations to be sent to the server and back to other clients. When comparing with the unsecured implementation of causal consistency, we can observe that the secure variants exhibit additional latency due to the use of cryptography and communication with the TiS.

Impact of server and TiS location: Figure 7b shows the effect of server and TiS location. We considered three server locations: local to G5k (in Grenoble), in Ireland, and in the US (east). For servers in AWS locations, latency of Client-Server increases as the latency to the data centers increases, as expected. For the Grenoble server, latency becomes better than for the Extended Secure model using the TiS at AWS locations, as obtaining the timestamp from the TiS is more expensive than sending the operation to the Grenoble server (located close to the center of the G5k network). With a TiS server in Grenoble (*S. Extended Causal - Grenoble*), the latency of the Extended Secure model became slightly better than that of Client-Server with a Grenoble server, as it is more efficient to propagate messages in an overlay network than having a server sending all messages to all clients.



(a) Variants of the application. (b) Server and TiS deployments. (c) Locality effect.

Figure 7: Client-client delivery latency (ms) with clients spread over Grid5000.

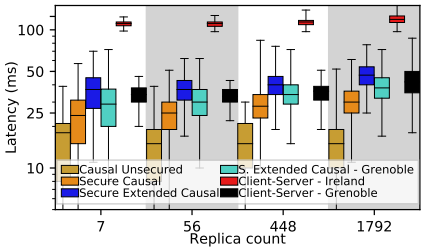


Figure 8: Latency results when varying the number of clients.

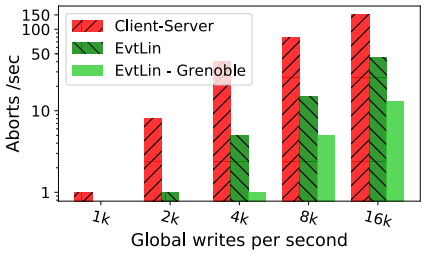


Figure 9: Effect of operating on stale views using EvtLin.

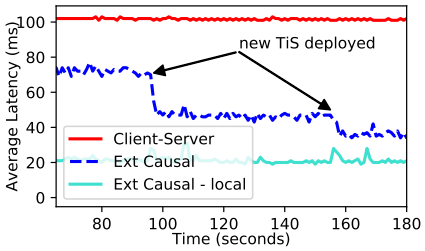


Figure 10: Latency when adding TiS servers (mean value of 1s windows).

Due to the same reason, the Secure Causal model has better latency than Client-Server with the server in Grenoble. The Secure Causal model is the only one that is not impacted by the latency to the server/service, as operations are propagated through the overlay (in the normal case).

Impact of data-locality: Figure 7c shows the difference of latency for operations received from nearby clients (running in the same DC) and from remote locations. As expected, the Client-Server solution shows no difference, as operations always have to be propagated through the server, even for operations from nearby clients. For secure consistency models (and Causal unsecured), there is a noticeable difference between operations from local and remote clients, which results from the underlying latency among clients. Secure Causal Consistency has considerably lower latency than other variants, for which the latency depends on the latency to reach the closest TiS instance to generate operations.

Discussion: The results suggest that the proposed decentralized secure consistency models are preferable to traditional cloud-based solutions when the latency among clients is lower than the client-server-client latency. We expect that this is the common case in cloud-based deployments, where an application is deployed in a small number of data centers.

When servers can be deployed close to clients, leading to client-server latency lower than the latency between distant clients, the decentralized models still exhibit considerable advantage for interactions among nearby clients. This makes the proposed models attractive for applications dominated by interactions among nearby clients, such as location-dependent information sharing and augmented reality games.

If an application is dominated by interactions among distant clients, and needs to use a consistency model stronger than Secure Causal Consistency, the advantage of the decentralized models is reduced (and depends on the location of TiS servers). In this case, the additional complexity of the proposed models might not be worth the benefit. We note that the cost and overhead of deploying and maintaining a fleet of full application servers is much higher when compared with TiS instances, which should also be taken into consideration when deciding which approach to adopt.

6.3 Scalability

Figure 8 reports latency between clients as a function of the total number of clients. The results show that, unlike the Causal unsecured model, the penalty in latency grows modestly with an increasing number of clients for all secure consistency models and for the Client-Server solution. The reason for this lies on the processing overhead in the replicas (cryptography) for the decentralized models and in the server (message propagation) for Client-Server. We have not scaled beyond 1792 clients due to lack of available hardware.

6.4 On Data Staleness

Games typically use some form of extrapolation to show expected object positions based on current object movement, but this can lead to objects jumping when the extrapolated value is stale due to actions of other players (which did not arrive before calculating expected positions). Decisions are thus made (by the players) on stale data. In this experiment we measure the effect of data staleness (due to propagation latency) on EvtLin and Client-Server baseline.

To increase contention, we used the game application but reduced the play-area and enabled player collisions. When clients operate over local stale data they possibly grab a coin concurrently with other players. In Client-Server, only the first grab operation to arrive at the server is accepted, with all concurrent operations being aborted. In EvtLin, concurrently applied (local at clients) operations must be undone and reapplied in the correct order (undo/redo).

We vary the number of operations per second to tune the amount of conflicts that occur. The results, in Figure 9, show that with few operations, the contention among players grabbing the coin is low (few aborts in Client-Server). As contention increases, due to the increasing number of operations per second, staleness increases which leads to additional aborts. EvtLin has significantly less aborts as clients operate over fresher data when compared with Client-Server.

This is confirmed by the even lower number of aborts in *EvtLin-Grenoble*, which, as discussed in the previous section, further reduces the latency to propagate operations.

The take away from these experiments is that enforcing application invariants that require coordination, with a low number of aborts, requires a scheme that lowers propagation latency of updates among clients, avoiding data staleness.

6.5 Impact of TiS deployment

Figure 10 shows the effect on latency of adding TiS instances closer to clients. Clients are scattered throughout the G5k clusters and continuously issue operations. Initially there is a single TiS instance in AWS EC2 Frankfurt. At the 90 and 150 second marks we deploy additional instances in AWS EC2 Paris and in G5k Grenoble, respectively. We also report the average latency for Client-Server (with the server in AWS EC2 Ireland) and Extended Causal with a TiS instance deployed at each G5k cluster (*Ext Causal - local*).

The results show that adding new TiS instances has a significant positive effect in the latency experienced by clients. This effect is more noticeable when the TiS is closer to clients. This process can be done in an autonomic fashion by, for instance, leveraging on work on database management on auto-tuning and commissioning replicas [41, 54].

The results also show the benefit of placing TiS servers very close to the clients. As TiS servers are lightweight, it is easy to deploy them in the edge of the network.

6.6 Impact of rational and arbitrary behaviour

We now discuss how client misbehaviour affects latency.

Discoverable (Malicious) behaviour: Figure 11a reports the latency observed by correct and incorrect clients in a scenario with multiple malicious clients, for which a proof-of-misbehaviour can be produced. In this experiment, incorrect clients (one third of all clients) follow the protocol up to the 27s mark, at which point they start propagating incorrect messages (trash, incorrectly signed, and tampered contents). This behaviour makes correct clients disconnect from such misbehaving clients. The latency perceived by incorrect clients grows beyond the round-trip-time to the server (*Server RTT*), as incorrect client need to communicate through the server. Correct clients continue experiencing low latency, as they restrict peer-to-peer interactions among them. Our experiments in varying the amount of malicious clients (5% to 95%) all presented similar results.

Undiscoverable (Rational) behaviour: Rational clients can delay operation propagation to other clients or the server, while being fast to disseminate to other rational clients (i.e., a form of collusion). We focus on delaying propagation, as other attacks either have the same or smaller effect, or lead the rational clients to be discovered. Figure 11b reports the latency with an increasing fraction of rational clients colluding to delay messages sent to correct clients. The results show that rational clients always observe operations of correct clients fast (*Inc-Corr*) and correct replicas observe operations from rational clients with a high delay (*Corr-Inc*).

Interestingly, correct clients start to perceive a noticeable effect among operations generated by themselves (*Corr-Corr*) when the amount of colluding rational clients grows over 50%. Similarly, at that point, the latency perceived by rational clients for operations issued by other rational clients (*Inc-Inc*) significantly drops. This is related with the probability of a correct (resp. rational) client having another correct (resp. rational) client as a direct neighbour, which depends on the fraction of rational clients, as neighbours are mostly selected at random. This implies that in our system, correct clients will benefit the most as long as they are the majority of participants (this was observed already in [50]).

Attacking the speculative execution of EvtLin: As discussed previously (Figure 9), EvtLin is significantly impacted by local data staleness when creating new operations. Delayed operations can lead to increased data staleness and force frequent undo/redos. We designed an experiment to measure the capacity of rational clients to disrupt a system using EvtLin by delaying operation propagation, where clients manipulate a bounded counter concurrently. Figure 11c shows the moment and number of operations undone/reapplied at each client. As baseline, we show the results when no client delays propagation (*NoCheat*) – in this case, contention over the counter already leads to some undo/redos. For disrupting the system, a rational client obtains from the TiS timestamps for 5 operations (identified by *Create* at the x axis) propagating these operations much later (at the following *Release*). This leads to a high number of undo/redos (*Cheat*), as operations with later timestamps have to be undone/reapplied by each client.

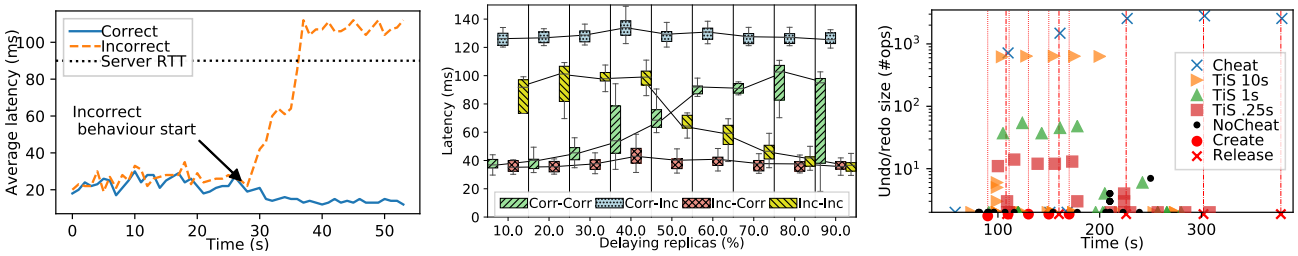
We also investigated if the problem could be addressed by having TiS instances storing timestamped operations. In this case, clients poll the TiS for operations that they might be missing – we use a polling time of 10s, 1s, or 0.25s. The late release of an operation does not lead to a large number of undo/redos, as clients obtain the delayed operations directly from the TiS service, showing that keeping operations at the TiS mitigates this attack. Naturally, enabling clients to poll the TiS more frequently increases the mitigation effect at the expense of additional resource consumption.

Discussion: It is clear that rational replicas cannot be removed altogether – users can always communicate out-of-band to gain a latency benefit. This is also true in the client-server model, as nothing disallows such replicas from running additional software. Our approach in reducing inter-replica latency mitigates this by letting correct clients observe operations from other correct clients with low latency.

6.7 Discussion on Network Performance

When running the experiments, we also measured the network usage and concluded that, when using our system, the server makes a lower use of the network (about one third) when compared with the classic Client-Server model. This happens because our system elects a fraction of clients to maintain communication with the server [34, 86], having the remaining clients only interacting with other clients. This enables the server to support a larger number of clients.

Comparing secure and non-secure consistency variants, there is an overhead when a client bootstraps since cryptographic keys must be generated and exchanged with the server to obtain a lease – this is a one-time cost. The overhead on the network due to secure mechanisms among clients depends on the application's message sizes. Experiments using small messages between clients incur an increase of bandwidth usage of up to 2.5×, due to the additional signatures and timestamps, which can have a considerable size for small data sizes (256 bytes for signatures with key sizes of 2048 bytes). This additional cost tends to ×1 (minimal overhead) when the size of application messages increases.



(a) Discoverable incorrect behaviour. (b) Rational effects on latency by delaying. (c) Effect on EvtLin undo/redo sizes as rational clients hold back operations.

Figure 11: Effect of rational and malicious behaviour.

7. RELATED WORK

Consistency: Web applications are typically client-server architectures, leading to high user-to-user latency due to client-server-client propagation [19, 20, 81, 84]. Weak consistency models [3, 12, 22, 24, 68, 73] can be used to reduce latency, requiring mechanisms to merge concurrent updates. We leverage CRDTs [77] to provide eventual convergence.

Byzantine fault tolerance (BFT) is designed to cope with malicious nodes that can perform arbitrary actions [16, 45]. In the context of replicated systems, solutions typically focus on the strong-consistency model among limited numbers of replicas [15, 16, 74]. Secure causal BFT [30, 75, 31] requires the total order established in BFT to respect causal order. Our work focus on weak consistency models, requiring no total order on operations. This leads to different requirements and properties for the proposed models and algorithms.

Blockchain approaches [4, 14, 33, 82] enforce a total order but the decentralized consensus algorithms used (e.g., proof-of-work) impose high latency on writes. These algorithms are sensitive to a majority of replicas acting against the rest. Our system mitigates the effects of a majority of incorrect nodes by using a trusted fallback – the central server.

Bayou [85] uses a form of eventual linearizability by relying on a primary server to assign a global order of all operations. An operation is tentatively executed in a replica until its global order is known. Our implementation of eventual linearizability orders operations based on the timestamps assigned by the TiS service. Furthermore, our algorithm provides protection against malicious replicas.

Peer-to-peer Middleware: Leveraging weak consistency together with peer-to-peer communication offers lower latency and reduced server load compared to using a client-server communication model [49, 52, 55, 56, 86]. Most peer-to-peer content delivery schemes [52, 55, 56] only allow clients to share static content and do not cope with simple man-in-middle attacks to provide malicious or fake content.

Atum [39] offers resilient group communication for large dynamic networks, but does not address attempts to subvert the system itself. FlightPath [49] is a peer-to-peer data streaming system tolerating up to 10% of malicious nodes, with the remaining peers acting rationally. S-Fireflies [26] is a data dissemination system robust to Byzantine faults. In contrast to our system, it relies on global knowledge of all nodes at all times. BAR Gossip [50] proposes algorithms for reliable dissemination in the BAR model using peer-to-peer gossip protocols. Our work is complementary to these works and could use the proposed solutions to achieve reliable dissemination (instead of leveraging a centralized component).

Legion [86] middleware offers a peer-to-peer communication mechanism with bolted-on causal delivery. Although using secure channels and providing access control, Legion does not deal with node misbehaviours after joining the system. In contrast, we focus on dealing with nodes attempting to misbehave by circumventing the data-layer's properties.

Secure hardware: Trusted Execution Environments can be used at the client to provide additional security guarantees for user-executed code [61, 70, 8]. Using trusted hardware at each client can provide a secure execution environment for a given application or web-page [37] or even disallow un-authorized users from accessing sensitive information [9]. Although in §5.3 we propose using such hardware for Secure Strict Causal, such modules are not widely available and recent works [13, 51] show possible attacks. These solutions do not consider attacks based on delays or collusion, which we partially address with the timestamping service.

8. CONCLUSION

This paper addresses the impact of replica misbehaviour on the guarantees of the causal consistency model. We analyzed the possible replica misbehaviours, derived secure consistency models that prevent different types of misbehaviour, and proposed algorithms for implementing these models. Depending on the application, a client may gain an unfair advantage by executing a different type of misbehaviour. By using the secure model that prevents such misbehaviour, an application may prevent such attacks.

Our evaluation shows that the proposed algorithms to enforce secure consistency models impose a modest overhead when compared with unsecured versions, while keeping much lower user-to-user latency and a reduced server load when compared to client-server solutions. Additionally, our algorithms effectively mitigate the effects of incorrect clients (even when colluding). Preventing different types of misbehaviour leads to different overheads, justifying the interest of providing different consistency models.

Acknowledgments: We thank the anonymous reviewers for their comments that helped improving the paper. This work was partially supported by FCT / MCTES grants PTDC/CCI-INF/32662/2017, Lisboa - 01 - 0145 - FEDER - 032662 (SAMOA), UIDB/04516/2020 (NOVA LINCS), and SFRH/BD/117446/2016 and by the AWS Cloud Credits for Research program. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see www.grid5000.fr).

9. REFERENCES

[1] S. Alagar. Causally ordered message delivery in mobile systems. In *1994 First Workshop on Mobile Computing Systems and Applications*, pages 169–175. IEEE, 1994.

[2] S. Alagar and S. Venkatesan. Causal ordering in distributed mobile systems. *IEEE Transactions on Computers*, 46(3):353–361, 1997.

[3] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[5] A. ANSI. X9. 95 standard for trusted time stamps, 2012.

[6] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[7] C. Baquero and N. Preguiça. Why logical clocks are easy. *Communications of the ACM*, 59(4):43–47, 2016.

[8] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to SGX. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 245–260. IEEE, 2016.

[9] E. Bauman and Z. Lin. A case for protecting computer games with SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, page 4. ACM, 2016.

[10] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 163–176, New York, NY, USA, 2008. ACM.

[11] M. Bravo and L. Rodrigues. Towards affordable externally consistent guarantees for geo-replicated systems. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, page 3. ACM, 2018.

[12] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 111–126. ACM, 2017.

[13] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, Aug. 2018. USENIX Association.

[14] C. Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.

[15] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[16] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[17] K.-T. Chen, P. Huang, and C.-L. Lei. How sensitive are online gamers to network quality? *Communications of the ACM*, 49(11):34–38, 2006.

[18] M. Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, 2005.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[21] R. Curtmola and C. Nita-Rotaru. Bsmr: Byzantine-resilient secure multicast routing in multihop wireless networks. *IEEE Transactions on Mobile Computing*, 8(4):445–459, 2008.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41-6, pages 205–220. ACM, 2007.

[23] O. Deutsch, N. R. Schiff, D. Dolev, and M. Schapira. Preventing (network) time travel with chronos. In *NDSS*, 2018.

[24] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal consistency and latency optimality: friend or foe? *PVLDB*, 11(11):1618–1632, 2018.

[25] DigitalBazaar. Forge. github.com/digitalbazaar/forge#rsa.

[26] D. Dolev, E. N. Hoch, and R. Van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *International Conference on Principles of Distributed Systems*, pages 343–357. Springer, 2007.

[27] J. Dong, R. Curtmola, and C. Nita-Rotaru. Secure network coding for wireless mesh networks: Threats, challenges, and directions. *Computer Communications*, 32(17):1790–1801, 2009.

[28] J. R. Douceur. The sybil attack. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[29] V. Drabkin, R. Friedman, and M. Segal. Efficient byzantine broadcast in wireless ad-hoc networks. In

2005 International Conference on Dependable Systems and Networks (DSN'05), pages 160–169, June 2005.

[30] S. Duan, M. K. Reiter, and H. Zhang. Secure causal atomic broadcast, revisited. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 61–72. IEEE, 2017.

[31] S. Duan, M. K. Reiter, and H. Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041. ACM, 2018.

[32] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer. Kronos: The design and implementation of an event ordering service. In *Proceedings of the Ninth European Conference on Computer Systems*, page 3. ACM, 2014.

[33] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation NSDI 16)*, pages 45–59, 2016.

[34] H. Garcia-Molina. Elections in a distributed computing system. *IEEE transactions on Computers*, pages 48–59, 1982.

[35] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, Apr. 2018. USENIX Association.

[36] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[37] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza. Trustjs: Trusted client-side execution of javascript. In *Proceedings of the 10th European Workshop on Systems Security*, page 7. ACM, 2017.

[38] N. W. Group. Internet X 509 public key infrastructure time-stamp protocol (tsp), 2001.

[39] R. Guerraoui, A.-M. Kermarrec, M. Pavlovic, and D.-A. Seredinschi. Atum: Scalable group communication using volatile groups. In *Proceedings of the 17th International Middleware Conference*, page 19. ACM, 2016.

[40] IEEE. Ieee std 1588 - standard for a precision clock synchronization protocol for networked measurement and control systems, 2002.

[41] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011.

[42] K. Jannes, B. Lagaisse, and W. Joosen. The web browser as distributed application server: Towards decentralized web applications in the edge. In *Proceedings of the 2Nd International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '19, pages 7–11, New York, NY, USA, 2019. ACM.

[43] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 14(2):8, 2007.

[44] S. Kashyap, C. Min, K. Kim, and T. Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 34:1–34:15, New York, NY, USA, 2018. ACM.

[45] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, pages 45–58. ACM, 2007.

[46] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[47] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[48] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467. ACM, 2016.

[49] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin. Flightpath: Obedience vs. choice in cooperative services. In *OSDI*, volume 8, pages 355–368, 2008.

[50] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 191–204. USENIX Association, 2006.

[51] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[52] Y. Liu, Y. Guo, and C. Liang. A survey on peer-to-peer video streaming systems. *Peer-to-peer Networking and Applications*, 1(1):18–28, 2008.

[53] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[54] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[55] X. Lou and K. Hwang. Collusive piracy prevention in p2p content delivery networks. *IEEE Transactions on Computers*, 58(7):970–983, 2009.

[56] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and tutorials*, 7(1-4):72–93, 2005.

[57] P. Mahajan, L. Alvisi, M. Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11:158, 2011.

[58] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):1–38, 2011.

[59] D. Malki and M. Reiter. A high-throughput secure reliable multicast protocol. In *Proceedings of the 9th*

IEEE Workshop on Computer Security Foundations, CSFW '96, pages 9–, Washington, DC, USA, 1996. IEEE Computer Society.

[60] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[61] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel® software guard extensions (intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 10. ACM, 2016.

[62] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[63] D. L. Mills and P.-H. Kamp. The nanokernel. Technical report, DELAWARE UNIV NEWARK, 2001.

[64] D. Mónica, J. Leitão, L. Rodrigues, and C. Ribeiro. Observable non-sybil quorums construction in one-hop wireless ad hoc networks. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 31–40, June 2010.

[65] Niantic. Ingress (video game). www.pokemongo.com.

[66] Niantic. Pokémon go (video game). www.ingress.com.

[67] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, May 1983.

[68] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 7:1–7:16, New York, NY, USA, 2015. ACM.

[69] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems (TOCS)*, 7(3):217–246, 1989.

[70] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), Jan. 2019.

[71] D. A. Popescu and A. W. Moore. Ptpmesh: Data center network latency measurements using ptp. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 73–79. IEEE, 2017.

[72] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 744–751. IEEE, 1996.

[73] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, pages 30–33, Oct 2014.

[74] H. V. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In J. H. Anderson, G. Prencipe, and R. Wattenhofer, editors, *Principles of Distributed Systems*, pages 88–102, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[75] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):986–1009, 1994.

[76] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 95–104, New York, NY, USA, 2010. ACM.

[77] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types, 2011.

[78] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[79] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[80] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 21. ACM, 2004.

[81] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[82] M. Swan. *Blockchain: Blueprint for a new economy.* "O'Reilly Media, Inc.", 2015.

[83] G. Tato, M. Bertier, E. Rivière, and C. Tedeschi. Sharelatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application. In *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets*, MECC'18, pages 8–15, New York, NY, USA, 2018. ACM.

[84] D. B. Terry. Replicated data management for mobile computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 3(1):1–94, 2008.

[85] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, volume 95, pages 172–182, 1995.

[86] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 283–292, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[87] A. van der Linde, J. Leitão, and N. Preguiça. Δ-crdts: Making Δ-crdts delta-based. In *Proceedings of the 2Nd*

Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC '16, pages 12:1–12:4, New York, NY, USA, 2016. ACM.

[88] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[89] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based access control for weakly consistent replication. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 293–306, New York, NY, USA, 2010. ACM.

[90] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 253–267, New York, NY, USA, 2003. ACM.

[91] L. Zhang, G. Ding, Q. Wu, Y. Zou, Z. Han, and J. Wang. Byzantine attack and defense in cognitive radio networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(3):1342–1363, 2015.