Access Control for Weakly Consistent Data Stores

Mathias Weber and Annette Bieniusa

University of Kaiserslautern, Germany
{m_weber, bieniusa}@cs.uni-kl.de

Abstract. Information systems have become distributed over large distance networks to serve an ever-increasing demand for fast data access at global scale. This trend lead to a growing popularity of NoSQL data stores as they provide better performance and response times in the distributed setting than standard relational databases. To achieve these properties, these data stores sacrifice consistency guarantees for the data stored in favor of availability and robustness under network failures and partitions. Consequently, it is more difficult to secure such systems against unauthorized data access without introducing performance bottlenecks. Due to the weak consistency guarantees, it became much harder to build access control systems coupled with the data store because the policies are replicated and can become inconsistent. Using a separate strongly consistent system to implement access control seems more feasible, but this architectural design adds additional complexity and results in performance loss and single points-of-failure.
In this paper, we outline the challenges when building access control systems for distributed information systems based on weakly consistent data stores. Based on a formal model, we present a solution that correctly applies access control policies. It guarantees convergence of policy modifications that are concurrently issued at different datastore replicas.

Keywords: access control, security, weak consistency

1 Introduction

Traditionally, information systems where built in a centralized fashion with a single replica of all data. Information systems today are distributed all over the globe to provide fast response times and low latency. However, the techniques for developing information systems has drastically changed in the last years. This change was triggered by new trends such as the Internet of Things and Industry 4.0 as well as the rapid growth of the Internet.

These demands have lead to an increasing popularity of weakly consistent data stores. The main focus of these data stores is availability and fast response times they achieve by sacrificing the strong consistency guaranties that are at the core of traditional relational database systems. One consequence of this sacrifice is that the semantics of data stores have become more complex because weaker consistency guaranties allow more interleaving of operations, which leads to surprising behavior developer might be unaware of. Recently[4, 5, 12, 13],

Replicated Data Types have gained a lot of interest, partly also in industry. But the question of how to implement access control on weakly consistent data stores is an open question.

Access control systems guarantee that every action performed adheres to a set of rules, which can be dynamically changed at runtime. In traditional systems, this guarantee can be enforced by relying on a central server. This server fixes a total order of the operations which avoids conflicts. Using such a centralized architecture is not possible in a highly available, globally distributed system that requires low latency. A central access control server introduces a severe bottleneck and increases the total latency of all actions in the system. To reduce the latency we can sacrifice part of the consistency guarantees offered by the access control system. Gilbert and Lynch [8] have shown that high availability, partition tolerance and strong consistency cannot be achieved by any distributed system at the same time. This theorem is also known as the CAP theorem. One possible solution is to not handle the policies by a central server, instead they are replicated to different servers. This introduces a security threat since the rules can be modified on different servers in non-consistent ways. The access restrictions are based on the local copy of these policies, which can be outdated.

We show the importance of the causal relation between data operations and access control operations for the correctness of an access control system (Section 3). We describe the design-space of access control systems for weakly consistent data stores with replication (Section 4). We created a formal model for such a system which works applicative and achieves convergence of the policies of different replicas and sketch the most important proofs of the model including the proof of eventual consistency (Section 5). The model is formalized in Isabelle/HOL.

2 Information System Model

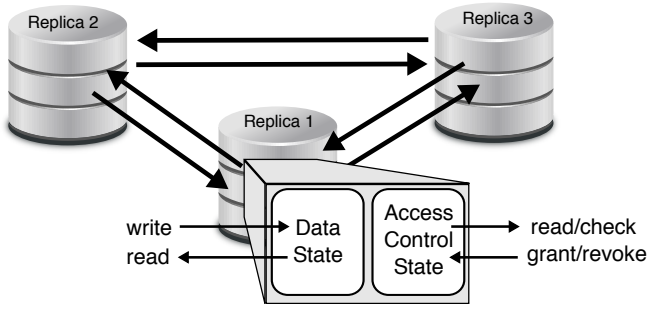

Fig. 1. Overview of the system model

We model an information system as a set of replicas where each replica represents a server with a local copy of the data managed by the information system.

Figure 1 shows an overview of the system. The replicated data consists of the data state and the access control state. The data state consists of the cooperative operations which make up the state of the part of the world represented in the information system. The state can be accessed using *read* and *write* operations. The access control state consists of access control policies, which influence the decision of the access control system. Policies in the access control state can be changed using the *grant* and *revoke* operations, which grant and revoke rights of a user to read and modify the data state and to grant or revoke rights of other users. In addition, the current set of policies can be inspected and system operations can be checked for compliance with the current policies using the *check* operation. Each replica has a local access control system allowing or disallowing execution of local operations on the replica. The decision of the access control system is only based on the local copy of the access control state. We assume the level of rights to be the level of replicas which means in order to restrict the rights of individual users each user has to work on his/her own replica. A generalization of this model to individual users and more complex access control policy patterns such as groups is left as future work.

Operations performed by a replica are broad-cast to the other replicas using messages. Each messages can be uniquely identified and caries only a single data or access control operation. Sending and receiving messages is asynchronous, so the sending replica does not wait for the message to be accepted by all replicas. To simplify things we assume reliable transport of messages, which means message loss has to be compensated on the network level, and full replication, that is every replica has a full copy of the data available.

3 Causal Consistency

To illustrate the importance of the causality of operations we start with an example system. Consider a social network where users can create a personal page, galleries for uploaded pictures and it is possible to invite friends to look at your personal data. By default, the system denies access to your personal page to all users. Other users can be added to a friend list. Users on this friend list have full access to all personal data including the posts on the personal page as well as all galleries.

Anne has an existing list of friends. One of these friends is Paul. After a heated discussion with Paul, Anne removes him from her friend list before uploading her new photos of the last party. Anne does not want Paul to have access to her new photos after removing him as a friend.

There is a causal relation between the remove operation of Paul and the upload of the new photos. In a system with strong consistency guarantees, it would not be possible for Paul to access the new photos as long as he is not readded to the friend list of Anne. When considering a replicated system which does not retain the causal relation between operations, there might be a server that receives the upload of the photos before the update of the friend list. This

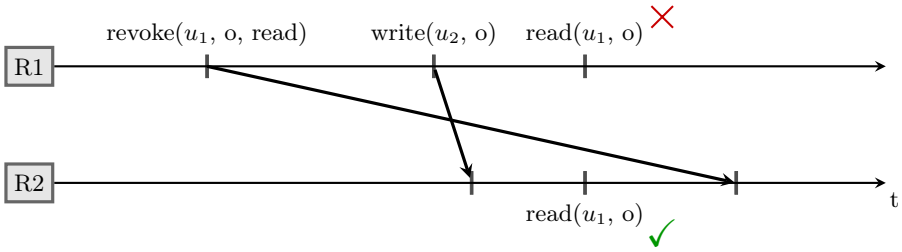

Fig. 2. Undesired semantics because of causality violation.

gives Paul the possibility to access the new photos before the change to the friend list of Anne is known to this local server.

As we see in the example above, it is important for the correctness of an access control system to preserve the causal relation between operations. We can distinguish several cases: (1) an operation on some data is performed because it is allowed by the current policies of the system; (2) the application computes new values based on the data state it sees in the system; and (3) the application changes the policies of the access control system based on previous policy changes.

Case (1) sketches the usual operation of the system: The data operations in the system are checked by the access control system to comply with the current policies. Only operations allowed by the access control policies may be performed by the system.

Case (2) describes the normal operation of the system without access control. Every computation reads values from the system, computes new values and enters them into the system. This relation between the values read and the values entered should be kept intact by the data store.

The last case (3) is how access control policies evolve. The initial state of the access control system consists of an initial set of policies. These policies can be adapted over time, for example because responsibilities and roles of persons change.

The causality between data operations can be loosened depending on the data types used and the guarantees needed by a specific application. Loosening the causality between access control policy changes and data operations invalidates the guarantees an access control system should offer.

4 Distributed Access Control

When designing an access control system for a distributed system there are some issues that cannot be avoided. We sketch these issues and their influence on the access control system.

As discussed in Section 3, the causal relation between access control policy changes and data operations have to be retained. The problem is illustrated in

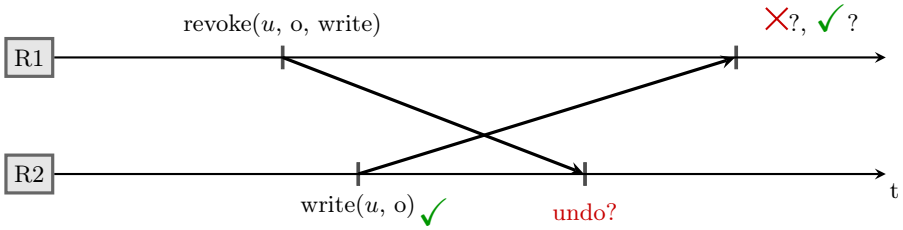

Fig. 3. Possible inconsistencies because of local decision.

Figure 2. There are existing data stores that offer causal consistency guaranties such as SwiftCloud [9] and Antidote [1]. Therefore, we can assume in our model of the access control system that the data store offers causal consistency as an option and will not go into the details of implementing causal consistency for the access control system. Even though the causal relation between access control operations and data operations is important, it is not sufficient for the convergence, and thus for the correctness of the access control system.

Since the access control state is local to the replica and synchronized using message broadcast, the access control state of different replica is temporarily inconsistent. Figure 3 shows such a scenario. Replica R1 performs an operation which revokes the right of user u to perform *write* operations on object o. This change of the access control policies is transmitted to replica R2 using a message. While this message was not yet accepted by R2, u tries to write to o, which is accepted by R2 according to the current local version of the policies. If the same operation would have been performed on R1, the access control system on R1 would have denied the operation. When the revoke message arrives at R2, it becomes clear that the access control state was inconsistent.

To avoid such inconsistencies, there are two possible solutions. Receiving the revoke operation on R2 could result in undoing of the *write* operation. This approach is favored by Cherif et al. [6] and Samarati et al. [10] and is known as optimistic approach. Alternatively, the *write* operation takes priority over the *revoke* operation, since the *revoke* was not yet accepted while executing the *write* on R2 and cannot easily be undone. This interprets the *write* operation as if it would have happened before the *revoke* operation. This approach needs access to the history of previous access control policies, since R1 has to check the validity of the *write* operation before accepting it[1].

One design decision to be made for a replicated access control system is whether to trust the other replicas or to put the trust only in the access control policies. When not trusting in the correctness and reliability of replicas, the broadcast messages have to be checked and accepted by the local access control system in order to protect the system from malicious actions. On the other hand, expecting the replicas to be distrusted makes it possible to place replicas in an

[1] We expect the data state to converge with the help of a different strategy such as convergent replicated data types

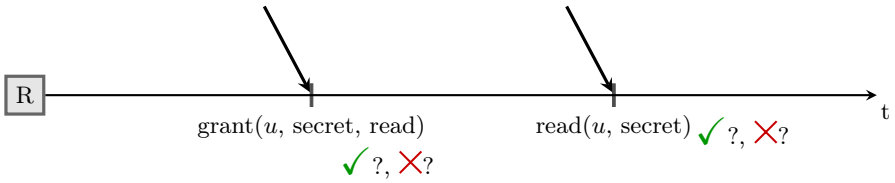

Fig. 4. Missing checks of remote messages.

untrusted environment such as the computer of a user. Figure 4 shows a simple attack scenario. An attacker u can run a manipulated replica which sends a forged message to replica R. Using this message, u can try to grant himself the right to read a secret object, even though u might not have the right to perform this policy change. If the message was accepted by R, this would lead to a set of access control policies which would grant a future *read* access to the *secret* object by u until the policies are repaired. This inconsistency is not acceptable with respect to the semantics of the access control system. Wobber et al. [14] describe a system that uses public and private key certificates to achieve trust in policies changes. A different way is to only accept policy changes that can be deduced to be allowed by the default policy set and already accepted changes.

The access control state has to converge on all replicas. Since the policies are changed locally and forwarded asynchronously, we cannot directly avoid inconsistent changes to the policies by different replicas. Other systems [6, 10] solve this problem by assigning one replica as the owner of an object. Only the owner replica can make the final decision about policy changes regarding the object owned. Since we want to support partitioning for the network, we have to follow a different direction. If, for example, we consider groups in social networks, the members of these groups may be distributed over the world and specific replicas may be separated because of network errors. Fixing a replica which may change the membership of a group would lead to unacceptable down-times.

If we accept that multiple replicas may change the access control policies for each object we also have to accept that there will be a window where the policies will not be consistent. The messages that distribute these changes to the access control state of remote replicas can arrive in any order because of the concurrent nature of the system and the delay caused by the network. Therefore we need a merge algorithm that incorporates remote changes into the local policies such that the outcome on all replicas is the same after processing all pending messages. We call this property the *convergence* of the access control state.

The naive approach of just accepting policy changes as they arrive at a replica can lead to different decisions on two replicas. Figure 5 shows the problem. The messages have to be extended by meta-data which allows to order the messages and this order is the same on all replicas.

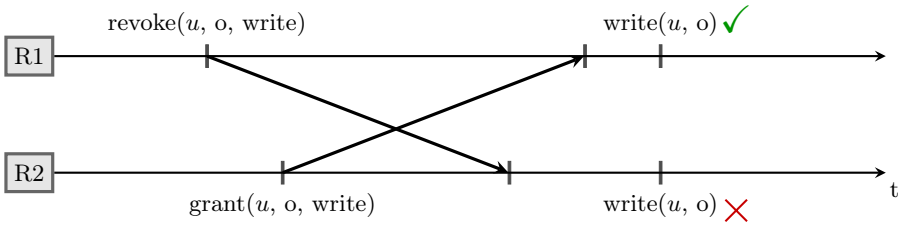

Fig. 5. Merging Policy Changes

5 Access Control Model for Distributed Information Systems

In the previous section we described the design space of implementing an access control system for globally distributed replicated information systems. In the following, we present a solution.

5.1 Formal Model of the Data Store

Replicas	$r \in R$
Data operations	$op_D \in Op_D = R \times OT \times \mathbb{N}$
Admin. operations	$op_{AC} \in Op_{AC} = AOT \times R \times OT \times \mathbb{N}$
Messsages	$msg \in M = R \times Op_D \cup Op_{AC}$
Global State	$gs \in R \times RS \times opDeps$
Replica state	$rs \in RS = \overline{Op_D} \times \mathcal{P}(Op_{AC}) \times \overline{M} \times \mathcal{P}(M)$
Dependencies	$opDeps :: M \mapsto \mathcal{P}(M)$
Operation Type	$ot \in OT$
Adm. Context	$actx \in \mathbb{N}$
Adm. Op.	$aot \in AOT = \{Grant, Revoke\}$

Fig. 6. Definitions used in the formal model

The over-bars symbolize sequences. For example $ms \in \overline{M}$ stands for a sequence of messages where the ith element can be accessed by $ms@i$.

We assume the set of replicas to be fixed, meaning neither can new replicas be added to the system nor can existing replicas be removed. Further, we simplify the model by assuming that each user works on his/her own replica.

State The global *data store* state gs consists of a set of replica (replicas(gs)), their local states (replicaState(gs)) as well as the operation dependencies (called operationDeps(gs)). The *operation dependencies* is a mapping from a message m to the set of messages *deps* that m causally depends on. The *local state* of a replica consists of the data state (persistentOps(ls)), in form of a list of data operations that have been performed on the replica, and the access control state ($admOps(ls)$), in form of a set of administrative operations or policy changes performed on the replica. In addition to that, the local state also consists of a list of accepted message (acceptedMessages(ls)) and an incoming queue (incomingQueue(ls)), the queue of messages that have been sent by other replicas but were not yet accepted by the replica. The incoming queue is modeled as a set to reflect the non-determinism involved in transferring messages over a switching network. In this way the order in which the broadcast messages are handled by each replica is not fixed by the incoming queue.

Operations A *data operation* $op_D = (r, ot, actx)$ consists of the origin replica r the operation was first performed on, the operation type ot, which can also include the target object of the operation, and the administrative context $actx$. The administrative context is needed by the access control system to determine the policy that allowed this operation on the origin replica. An *administrative operation* $op_{AC} = (aot, r, ot, actx)$ consists of the type of the administrative operation aot, either $Grant$ or $Revoke$, the replica r and operation type ot this policy change regulates and the administrative context. The messages distributing the changes to other replicas are *data messages* $msg_D = (r, op_D)$ and *administrative messages* $msg_{AC} = (r, op_{AC})$ and each consist of the sending replica and the operation to be transfered. We assume that each message can be uniquely identified and from the models point of view no message is sent or received twice.

Initial State When starting-up for the first time, the system is in its *initial state*, which means all replicas are in initial local state, there are no dependencies between messages and the policies are equal for all replicas. These initial policies can for example state that the root-user (in our case the root-replica) may assign rights to other users (replicas) and all other users have no rights at all. A replica is in *initial local state* if the data state and the access control state is empty and no messages have been accepted and the incoming queue is empty.

Access Control Policies Before we come to the steps such a system can make we first have to clarify what the access control policies are. The *default policies* apply when no other policy in admOps(ls) applies for the operation on the replica. The access control policies of a replica state is the set of default policies plus the set of administrative operations processed by a replica

$$\text{acPolicies}(ls) = \text{defaultPolicies} \cup \text{admOps}(ls)$$

Each administrative operation carries an administrative context which is an element of a totally ordered kind such as the natural numbers. This context is

like a version number for policy updates. The *relevant policy* for an operation on a replica relevantPolicy(r, ot, ls) is a policy $p = (aot_p, r_p, ot_p, actx_p)$ where $r_p = r$ and $ot_p = ot$ and

$$p \in \text{acPolicies}(ls) \wedge$$
$$\forall\ (aot'_p, r'_p, ot'_p, actx'_p) \in \text{acPolicies}(ls).\ (r'_p = r \wedge ot'_p = ot) \implies actx_p \geq actx'_p$$

In other words: The relevant policy is the policy in the set of the access control policies with the greatest access control context. A data message is *allowed to be sent* by a replica sendOK(ls, msg_D) where $msg_D = (r, op_D)$ and $op_D = (r, ot, actx)$ if the relevant policy for this message allows the operation in the current administrative context

$$\text{relevantPolicy}(r, ot, ls) = (Grant, r, ot, actx)$$

The operation is performed on r and broadcast to the other replicas.

A administrative message is *allowed to be sent* by a replica sendOK$(ls, msg_{AC}$ where $msg_{AC} = (r, op_{AC})$ and $op_{AC} = (aot, r, ot, actx)$ if the relevant policy is the inverse of the operation

$$\text{relevantPolicy}(r, ot, ls) = (\neg aot, r, ot, actx - 1)$$

The inverse of $Grant$ is $Revoke$, $\neg Grant = Revoke$ and the other way round $\neg Revoke = Grant$.

The *allowing policy* for a data message allowingPolicy(msg_D) where $msg_D = (r, op_D)$ and $op_D = (r, ot, actx)$ is the corresponding policy $(Grant, r, ot, actx)$. r in this case is the replica which originally executed the operation first and also the sender of the corresponding message. The *allowing policy* for an administrative messages allowingPolicy(msg_{AC}) where $msg_{AC} = (r, op_{AC})$ and $op_{AC} = (aot, R, ot, actx)$ is $(\neg aot, r, ot, actx - 1)$. This means an administrative operation may only be performed if the inverse policy with the previous administrative context is already part of the current policy set. This construction enforces that the administrative context is monotonically increasing for each policy change and therefore makes each relevant policy unique. A message is *allowed to be processed* processOK(ls, msg) if the allowing policy is in the local access control policy set allowingPolicy$(msg) \in$ acPolicies(ls).

We expect the data store to be causally consistent. A data store is *causally consistent* if for all messages accepted by a replica all dependencies have been accepted before. We write $l@i$ to denote the ith element of list l.

$$\text{causallyConsistent}(gs) \equiv$$
$$\forall r \in \text{replicas}(gs), \forall rs = \text{replicaState}(gs, r), \forall m1, i.m1 = \text{acceptedMessages}(rs)@i,$$
$$\forall m2 \in \text{operationDeps}(gs, m1).\ \exists i' < i.\ \text{acceptedMessages}(rs)@i' = m2$$

A message is *causally ready* on a replica if all its dependencies have already been accepted.

$$\begin{aligned}&\text{causallyReady}(gs, msg, ls) \equiv \forall m \in \text{operationDeps}(gs, msg)\\&\quad \exists i.\, \text{acceptedMessages}(rs)@i = m\end{aligned}$$

Steps With these definitions we can define the possible steps how the system can evolve: When doing a *step from one global state to the next* $gs \longrightarrow gs'$, we can either accept a message from the incoming queue $gs \xrightarrow{accept} gs'$, or a replica can perform an operation and send a new broadcast message $gs \xrightarrow{send} gs'$. A message msg may only be accepted by a replica r with replica state rs if

$$\begin{aligned}&msg \in \text{incomingQueue}(rs) \wedge \text{causallyReady}(gs, msg, rs) \wedge\\&\text{acceptMessage}(msg, rs, rs') \wedge \text{processMessage}(msg, rs, rs')\end{aligned}$$

The message may only be *accepted* if it is causally ready and currently in the incoming queue. It is accepted by removing it from the incoming queue and appending it to the list of accepted messages of the replica.

$$\begin{aligned}&\text{acceptMessage}(msg, rs, rs') \equiv\\&\quad \text{acceptedMessages}(rs') = \text{acceptedMessages}(rs) + [msg] \wedge\\&\quad\quad \text{incomingQueue}(rs') = \text{incomingQueue}(rs) - \{msg\}\end{aligned}$$

The system *processes the message* only if the message is allowed to be processed. Otherwise the effect of the message is not visible on the receiving replica. If the message is allowed to be processed, the operation of a data message is added to the data state and the administrative operation of an administrative message if added to the set of access control policies.

$$\begin{aligned}&\text{processMessage}(msg_D, rs, rs') \equiv \text{processOK}(rs, msg_D) \implies\\&\quad \text{persistentOps}(rs') = \text{persistentOps}(rs) + [op_D]\end{aligned}$$

$$\begin{aligned}&\text{processMessage}(msg_{AC}, rs, rs') \equiv \text{processOK}(rs, msg_{AC}) \implies\\&\quad \text{acPolicies}(rs') = \text{acPolicies}(rs) \cup \{op_{AC}\}\end{aligned}$$

A message msg may only be sent by a replica r with replica state rs if

$$\begin{aligned}&\text{msgSender}(msg) = r \wedge\\&\text{sendOK}(rs, msg) \wedge\\&\text{broadcast}(gs, r, msg) \wedge\\&\text{acceptedMessages}(rs') = \text{acceptedMessages}(rs) + [msg] \wedge\\&\text{incomingQueue}(rs') = \text{incomingQueue}(rs) \wedge\\&\text{processMessage}(msg, rs, rs') \wedge\\&\text{operationDeps}(gs', msg) = \text{set}(\text{acceptedMessages}(rs))\end{aligned}$$

Replica r has to be registered as the sender of the message, so the message has the form $msg = (r, op)$ where op is either a data operation or an administrative operation. Message may only be sent and therefore operations on the replica only be performed if they are allowed by the local policies $\text{sendOK}(rs, msg)$. If the permission is granted, the message is locally accepted and processed and broadcast to the other replicas. *Broadcasting* in our case means to add the message to the incoming queue of all other replicas. The operation dependencies of the message are registered to be all the messages that have been accepted by the replica at the time of sending the message.

5.2 Properties of the Data Store

Next, we want to show that the system we have described using the above model is causally consistent. For this, we first show an intermediate lemma.

Lemma 1. *If the global state gs is causally consistent, message msg is causally ready on replica r, r accepts the message and the resulting state is gs', then gs' is also causally consistent.*

Proof. Because gs is causally consistent, we know that for each message m that has been accepted by each replica, the dependencies of m have already been accepted before m. Only r accepts a new message and for all other replicas the list of accepted messages stays the same and the operation dependencies are not changed between gs and gs'. So, regarding the other replicas that systems stays causally consistent. Accepting msg on r means appending msg to the end of the list accepted messages of r

$$\begin{aligned}&\text{acceptedMessages}(replicaState(gs', r)) = \\ &\quad \text{acceptedMessages}(\text{replicaState}(gs, r)) + [msg]\end{aligned}$$

For the prefix of the list that consists of the accepted messages in gs, we known, that all dependencies have already been accepted in this sublist. It remains to show that the newly added message msg does not break the causal consistency. Because we known that msg is causally ready, we also known that the operation dependencies of msg have already been accepted by r, which is the definition of causally ready. This also means that each message in the set of operation dependencies can be found in the list of accepted messages of r, so adding msg to the end of the list of accepted message does also not break causal consistency. Thus we can follow that gs' is also causal consistent. □

Next we can use Lemma 1 to show that steps on the system will not break causal consistency.

Lemma 2 (Causal Consistency Preservation). *If we start in a causal consistent state gs and do a step $gs \longrightarrow gs'$, then gs' is also causal consistent.*

Proof. Case 1 ($gs \xrightarrow{accept} gs'$ accept a message msg on replica r). We assumed gs to be causal consistent and we known from the definition of accepting a message that this message needs to be causally ready. It follows from Lemma 1 that accepting a causally ready message msg in a causal consistent state gs yields a causal consistent state gs'.

Case 2 ($gs \xrightarrow{send} gs'$ send a message msg by replica r). The operation dependencies are changed in gs', the dependencies of msg are set to all messages already accepted by r. So we would have to recheck all accepted messages for all replicas. But we known that all message can be uniquely identified and that msg is a fresh message. This means that none of the replicas have already accepted msg before. In addition, we known that the other replicas except for r do not directly accept msg, but instead have msg in their incoming queue first. From this, we can deduce that causal consistency cannot be broken for the replicas other than r.

In case of r we know, that the replica accepts msg which means we append msg to the end of the accepted messages of r.

$$\begin{aligned}&\text{acceptedMessages}(replicaState(gs', r)) = \\ &\quad \text{acceptedMessages}(\text{replicaState}(gs, r)) + [msg]\end{aligned}$$

We know that msg can not be the prefix of the accepted message that is equal to the previously accepted messages because it is a fresh message. Thus this prefix cannot break causal consistency. It is left to show that appending msg does not break causal consistency. We know that the operation dependencies of msg are set to the messages currectly accepted by r, $\text{acceptedMessages}(\text{replicaState}(gs, r))$. Now we have to show that each of these messages have already been accepted by r in gs, which is trivial to see. Thus we have shown that gs' is causal consistent. □

From Lemma 2 we can deduce the correctness of the whole system with respect to causal consistency.

Theorem 1. *The data store as described in our model is causally consistent.*

Proof. By induction on the evaluation steps. □

The last important concept is that of the messages known to a replica. For the list of messages accepted by a replica we can show that all messages are accepted only once per replica. This means we can treat the list of accepted messages as a set. This reinterpretation is done by the set(...) operator. The incomingQueue is already treated as a set to model non-determinism of message transport. The set of *messages known to a replica* is the set of messages accepted by the replica plus the set of messages in the incoming queue

$$\text{knownMessages}(ls) \equiv \text{set}(\text{acceptedMessages}(ls)) \cup \text{incomingQueue}(ls)$$

We can show that all replicas know the same messages by induction over the steps performed.

Lemma 3. *If all replicas known the same messages in gs and we do a step from gs to gs' $gs \to gs'$, then all replicas know the same messages in gs':*

$$\begin{aligned} \forall r1, r2 \in \text{replicas}(gs).\, &\text{knownMessages}(\text{replicaState}(gs, r1)) = \\ &\text{knownMessages}(\text{replicaState}(gs, r2)) \wedge \\ gs \to gs' \implies & \\ \forall r1, r2 \in \text{replicas}(gs').\, &\text{knownMessages}(\text{replicaState}(gs', r1)) = \\ &\text{knownMessages}(\text{replicaState}(gs', r2)) \end{aligned}$$

Proof. We again distinguish which kind of step can be made

Case 1 ($gs \xrightarrow{accept} gs'$ accept a message msg on replica r). The accepting replica r removes msg from the incoming queue and adds it to the accepted messages. The known messages stay the same. The known messages of the other replicas also stay the same since their state does not change. □

Case 2 ($gs \xrightarrow{send} gs'$ send a message msg on replica r). The messages msg is directly processed by r and added to the accepted messages. The other replicas get msg in their incoming queue. Overall, msg is added to the known messages of all replicas and therefore the known messages of all replicas are the same. □

5.3 Properties of the Access Control System

Based on the model of the data store we construct the model of the access control system. We show some interesting properties of the access control policies before proving the convergence of the access control state on all replicas.

The access control state is monotonically increasing, old policies are not removed and changes are done by adding the changed policies to the access control state. This can be seen by looking at the possible steps and how messages are accepted by replicas. These changed policies are considered before the old ones when looking for the relevantPolicy.

Using this monotonicity of the access control state, we can show that all known messages can be processed by the sender of the message.

Lemma 4. *All known messages $msg \in \text{knownMessages}(gs)$ are allowed to be processed by the original sender of the message.*

Proof. We show the property by induction over the steps starting in the initial state *is*.

Case 1 (Initial state). In the initial state none of the replicas have accepted any messages and the incoming queues are empty. This means that there are no known messages yet, so the property hold trivially.

The proofs for the steps can be split-up into two cases, accepting a message and sending a new message:

We assume state gs can be reached from the initial state is by arbitrarily many steps $is \rightarrow^* gs$ and that all known messages in gs can be processed by the sender on the message

$$\forall m \in \text{knownMessages}(gs).\, r = \text{sender}(m) \implies \text{processOK}(r, m)$$

Case 2 ($gs \xrightarrow{accept} gs'$ accept a message msg on replica r). We have to show that all known messages can be accepted by their sender in gs' after processing msg on r. By using that the access control state is monotonically increasing we know that the access control policies in gs are a subset of the access control policies in gs'. We have already seen in the proof of Lemma 3 that the set of known messages does not change during accepting a message. Being allowed to process a message means that the allowing policy is in the access control policies of the replica trying to process the message, in our case the original sender of the message. Since the known messages are the same and the access control policies of gs are a subset of the access control policies of gs' for all replicas that means that the sender is still allowed to process the message.

Case 3 ($gs \xrightarrow{send} gs'$ send a message msg by replica r). In this case have to distinguish the previously known messages knownMessages(gs) and msg, which is added as a new message knownMessages(gs') = knownMessages(gs) $\cup$ $\{msg\}$. We can use the same reasoning as in the accepting case to show that knownMessages(gs) can be processed by their sender in gs'. What is left to show is that msg can be processed by its sender. We know that the sender of msg is r and that msg is allowed to be sent by r. Hence, the allowing policy is the relevant policy in the access control policies of r for the operation performed, which means that the allowing policy is in the access control policies of r. Thus, the known messages in gs' are all allowed to be processed by their sender. □

Using the causal consistency of the data store, we can transfer the property of being able to accept the message to the receiver.

Theorem 2. *All messages are allowed to be processed by the receiving replica once the message is causally ready.*

Proof (sketch). The proof uses Lemma 4 which states that the sender is allowed to process the message. This also means that the policy allowing to process the message is available on the sending replica and therefore gets registered as a dependency of the message to be sent. When another replica r wants to accept message msg, it has to wait until the message is causally ready. Thus all dependencies of msg have been accepted by the replica before accepting msg itself. This makes sure that the message msg_{AC} carrying the policy change that allowed sending the msg has been accepted by r before accepting msg. We can show that the state of the replica is valid, meaning all accepted messages have

also been processed by the replica and the effects of the messages have been materialized in the state. Because in a valid state the message msg_{AC} has been processed before processing msg the allowing policy is part of the access control policies of r. This policy then allows processing msg so all messages are allowed to be processed by the receiving replica.

Theorem 2 can be used to show the convergence of the access control state. For the state to converge, no new messages may be sent and the pending messages have to be accepted meaning a system converges in a state where all known messages have been accepted. In Theorem 2 we have shown that all accepted messages are processed by the replicas. This in combination with the set-semantics of the access control state *leads to convergence*.

6 Conclusion

In this paper, we presented the design-space of access control systems for weakly consistent data stores. We showed a formal model of an access control system for causally consistent data stores. One of the main results is that the causally between data operations and access control operations is important for the correctness of the access control system. In addition, we have shown that an applicative model in contrast to the optimistic models proposed by Cherif et al. [6] and Samarati et al. [10] still works without undoing processed operations and still the policies of all replicas eventually converge to a common state.

These results are still rather theoretical and abstract. The next steps will be to develop an access control model inspired by popular models like role-based access control [7, 11] or an authorization logic [2, 3] based on the lower-level model we presented. An implementation of such as system will be based on Antidote [1], a causal-consistent data store developed by the SyncFree Project [2].

[2] https://syncfree.lip6.fr/

References

[1] SyncFree/antidote (Jul 2015), https://github.com/SyncFree/antidote

[2] Abadi, M.: Logic in access control. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings. pp. 228–233. IEEE Comput. Soc (2003)

[3] Bauer, L.: Access Control for the Web via Proof-Carrying Authorization. Ph.D. thesis, Princeton University (2003)

[4] Bieniusa, A., Zawirski, M., Preguiça, N.M., Shapiro, M.: An optimized conflict-free replicated set. arXiv.org (2012)

[5] Burckhardt, S.: Principles of Eventual Consistency. Foundations and Trends in Programming Languages 1(1-2), 1–150 (2014)

[6] Cherif, A., Imine, A., Rusinowitch, M.: Practical access control management for distributed collaborative editors. Pervasive and Mobile Computing 15, 62–86 (2014)

[7] Ferraiolo, D., Kuhn, R.: Role-Based Access Control. In: In 15th NIST-NCSC National Computer Security Conference. pp. 554–563 (1992)

[8] Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51 (2002)

[9] Preguica, N., Zawirski, M., Bieniusa, A., Duarte, S., Balegas, V., Baquero, C., Shapiro, M.: SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops. vol. abs/1310.3, pp. 30–33. IEEE (Oct 2014)

[10] Samarati, P., Ammann, P., Jajodia, S.: Maintaining Replicated Authorizations in Distributed Database Systems. Data Knowl. Eng. 18(1), 55–84 (1996)

[11] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. IEEE Computer 29(2), 38–47 (1996)

[12] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506 (Jan 2011)

[13] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems pp. 386–400 (2011)

[14] Wobber, T., Rodeheffer, T.L., Terry, D.B.: Policy-based access control for weakly consistent replication. In: Morin, C., Muller, G. (eds.) European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010. pp. 293–306. ACM (2010)