A Contract-extended Push-Pull-Clone Model

Hien Thi Thu Truong, Claudia-Lavinia Ignat
INRIA Nancy-Grand Est, France
{hien.truong, claudia.ignat}@inria.fr

Mohamed-Rafik Bouguelia
Nancy University, France
mohamed.bouguelia@loria.fr

Pascal Molli
Nantes University, France
pascal.molli@acm.org

Abstract—**In the push-pull-clone collaborative editing model widely used in distributed version control systems users replicate shared data, modify it and redistribute modified versions of this data without the need of a central authority. However, in this model no usage restriction mechanism is proposed to control what users can do with the data after it has been released to them. In this paper we extended the push-pull-clone model with contracts that express usage restrictions and that are checked a posteriori by users when they receive the modified data. We propose a merging algorithm that deals not only with modifications on data but also with contracts. A log-auditing protocol is used to detect users who do not respect contracts and to adjust user trust levels. Our proposed contract-based model has been implemented and evaluated by using PeerSim simulator.**

Index Terms—**collaborative editing, contract-based model, push-pull-clone model, peer-to-peer computing, trust, log auditing.**

I. Introduction

Most platforms hosting social services such as Facebook or Google+ rely on a central authority and place personal information in the hands of a single large corporation which is a perceived privacy threat. Users must provide and store their data to vendors of these services and have to trust that they will preserve privacy of their data, but they have little control over the usage of their data after sharing it with other users. Several solutions were proposed to replace the central authority-based collaboration with a distributed collaboration that offers support for decentralization of services such as peer-to-peer file sharing (e.g eDonkey and BitTorrent), peer-to-peer applications for audio and video calls (e.g. Skype) and peer-to-peer social networks (e.g. Diaspora and Peerson). In the domain of collaborative editing of shared documents Distributed Version Control Systems (DVCS) such as Git [10] and Mercurial [15] are well-known examples of social software that demonstrate that it is possible to share data without the need for a collaboration provider. In DVCS systems users replicate shared data, modify it and redistribute modified versions of this data by using the primitives push, pull and clone. Users clone shared data and maintain in a local workspace this data and modifications done on this data. Users can then push their changes to different channels and other users that have granted rights may pull these changes from these channels. Throughout this paper we will refer to this model of collaboration as push-pull-clone model (PPC).

In the PPC collaboration model it is very difficult to control what users will do with the data after it has been released to them and that they will not misbehave and violate usage policy. Usage control mechanisms model obligation, permission and forbiddance as contracts [16] that users receive together with data which refer to what happens to data after it has been released to authorized people, for example, how they may, should and should not use it. The main issue addressed by this paper is how usage restriction can be expressed and checked within PPC model and what response actions can be taken in order to discourage/penalize behavior of users that misbehaved.

At the beginning DVCS systems were mainly used by developers in open-source code projects but nowadays they started to be widely adopted by companies for code development. In open source projects, the usage restriction is expressed in the license of the code, while in closed source code projects, it is expressed in the contracts developers sign when accepting their job. In both cases, usage restrictions are checked a posteriori outside the collaborative environment with social control or plagiarism detection. As a result of observations concerning usage violation trust on the users that misbehaved is implicitly decremented and collaboration with those users risks to be stopped. We aim at proposing a contract model that expresses usage restrictions as policies that can be checked within the collaborative environment.

Access control mechanisms do not address the issue of usage restriction after data was released to users. Traditional access control mechanisms prevent users from accessing to data and granted rights are checked before access is allowed. It has been shown that these access control mechanisms are too strict [4]. There exist some optimistic access control approaches [20] that check a-posteriori access policies. In these approaches, if user actions violate granted rights, a recovery mechanism is applied and all carried-out operations are removed. Usually, this recovery mechanism requires a centralized authority that ensures that the recovery is taken by the whole system. However, the recovery mechanism is difficult to be applied in decentralized systems such as DVCS where a user does not have knowledge of the global network of collaboration. Generally, access control mechanisms aim at ensuring that systems are used correctly by authorized users with authorized actions. Rather than ensuring a strong system security, we aim at offering a flexible approach based on contracts that can be checked after users gained access to data and on trust management mechanisms that help users collaborate with other users they trust. However, contract-based models such as Hippocratic databases [1] and P3P [6] have not been deployed for the push-pull-clone collaboration.

The main issue in designing a contract-based distributed collaboration is that contracts are objects that are part of the replication mechanism. In our contract-based model each user maintains a local workspace that contains local data as well as modifications done on the shared data and contracts related to the usage of the shared data. The logged changes and contracts are shared with other users. The merging algorithm has to deal not only with merging modifications on data but also with contracts. Moreover, conflicts have to be resolved not only between modifications, but also between contracts. For checking if users respect usage restriction, each user performs a log-auditing mechanism. According to auditing results users adjust their trust levels. To our knowledge there is no existing collaborative editing model based on contracts which allows to audit and update trust levels during editing process. Major contributions of this paper are as follows:

- A model of distributed collaborative editing based on contracts in weakly consistent replication that we call throughout this paper the C-PPC (contract extended push-pull-clone) model. The proposed model ensures CCI consistency [21] on the shared document.
- A set of experiments for evaluation of the performance of the C-PPC model and of the log auditing mechanism for detection of users misbehavior and for updating trust levels by using a peer-to-peer simulator.

The paper is structured as follows. We start by presenting an overview of our proposed approach. We then describe editing model and collaborative process which ensure properties of model and of contract specification. We next discuss the consistency of the model. We then briefly describe our solution to assess trust. We also provide some experimental results of the simulation to evaluate the efficiency of our model. We compare our work with some related approaches. We end our paper with some concluding remarks and directions for future work.

II. Overview of Our Approach

In this section we present an overview of our C-PPC model. Collaborative editing requires high levels of respect and trust among users. To create a trustful and respectful collaborative environment, we consider creating a collaboration contract that will be used in all collaborative interactions.

Let us give a simple example for illustrating how C-PPC model works. Let us consider a network composed of four users A, B, C, and D who trust each other. Users share and edit collaboratively a document. At the beginning the network is built based on social trust between users and connections are established only between users who trust each other. Users trust their collaborators with different trust levels that are updated according to their collaboration experience. For instance, A trusts both B and C, however, A has different trust levels for B and C and thus gives them different contracts over the shared document. For example, A gives B the permission to edit, while he/she gives C only the permission to view the document. Also, A gives C an obligation to send feedback while this obligation is not given to B. Receivers are expected to follow these contracts. Otherwise, their trust levels will be adjusted once misbehavior is detected. We log user modifications on the shared documents as well as contracts users give to other users when they share their modifications.

Our editing model uses push, pull and clone as native direct pair-wise communications between users. To work with others a user simply sets up a local workspace for his own work, and uses trusted channels to *push* his documents to trusted friends. Other users can then get the document by cloning (executing a *clone* primitive) the document from the user's workspace. In this way they have independent local workspaces for the shared document, do their changes locally and publish them by executing a *push* primitive. The user then executes a *pull* primitive to get the changes into his local workspace. Push, pull, and clone primitives are used for efficient distributed collaboration and they were already implemented in distributed version control systems such as Git and Mercurial. We assume the system uses a pairwise FIFO channel between two users for changes propagation to guarantee that messages are received in the order they were sent. This order can be preserved by using logical timestamps to sort messages into chronological order. Push, pull and clone communication primitives are operated on such ordered channel.

In Fig. 1, we present a scenario of our C-PPC model. Without trust management, users might share and reconcile replicas with all group members without taking into account if they well-behave or misbehave. In our C-PPC model users are not uniformly trusted and a user generally collaborates only with trusted users. Our model is adaptable to the case when a user changes his behavior from good to bad as his trust or reputation is reduced and other users will cease communication with that user. Users trust levels are adjusted mainly based on their past behavior. A misbehavior detection mechanism and trust model to manage trust levels of collaborators are mandatory to maintain correctly such network. In the Fig.1, the upper half shows the trust levels between users A, B, C and D; and the bottom half shows a collaboration scenario using our C-PPC model where users work on the shared document *Doc*. User D wants to share his changes with user B with the contract *forbiddance to delete* and *permission to share* and therefore he pushes his changes together with this contract to the channel of the communication with user B. As it is the first time that user B initiates a communication with user D he has to clone the repository with changes from user D. In parallel, user D collaborates with user C. We suppose that afterward, user C continues to collaborate with user B. As a result of log auditing of certain communications, user B discovers that C did not respect the contracts he received, then user B decrements the trust level of user C and cuts further collaboration with user C since his trust level becomes low. Further, user B does some changes that he pushes to user A with contract *forbiddance to delete* and *permission to share*. We can see from the figure that at the end user B stopped collaboration with users C that misbehaved and continues collaboration with user A that well behaved.

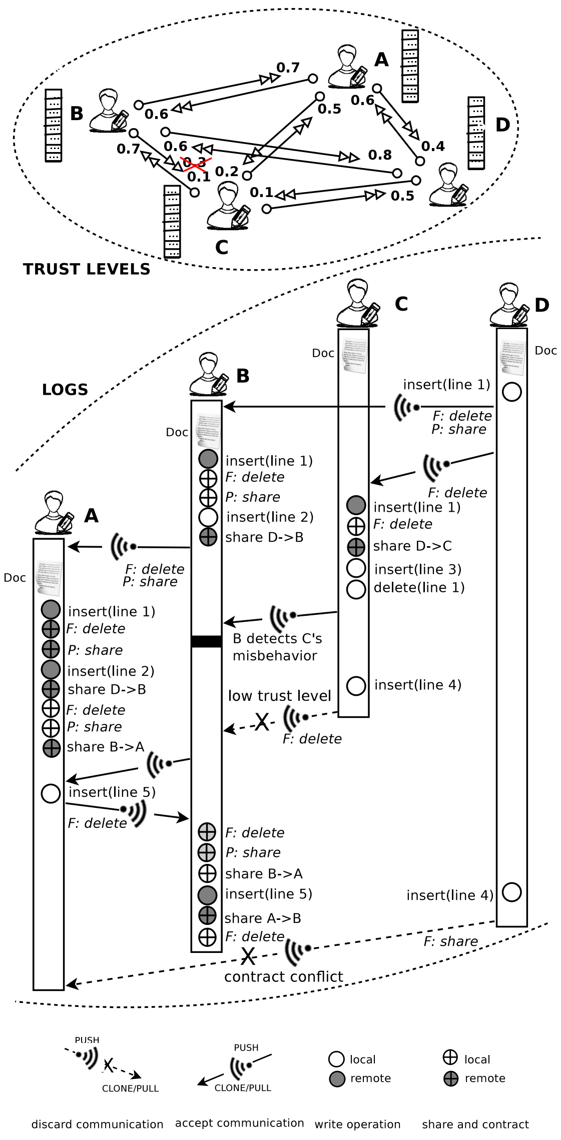


Fig. 1. Contract-extended PPC model. Users have different trust levels on others. They discard all communications from/to distrustful users.

III. Editing Model

In this section we describe our editing model composed of users, logs of operations on shared document and contracts between users.

A. Users

Users are main participants in our C-PPC model. Users are connected in a collaborative network based on the trust they have in other users. However users are not uniformly trusted by others. Each user keeps at his local site the replica of shared documents and works locally on these replicas. For sake of simplicity we consider that all users (also called sites throughout this paper) are collaborating on a single shared document.

B. Document, Changes and Logs

The system keeps a document as a log of operations that have been done during the collaborative editing process. The log maintains information about user contributions to different parts of the document and when these contributions were performed. The outcome of collaborative editing is a document that could be obtained by replaying the *write* operations such as *insert*, *delete*, *update* from the log. Two users can make progress independently on the shared document. Changes are propagated in weakly consistent manner, i.e. a user can decide when, with whom and what data is sent and synchronized. Push, pull and clone communication primitives are operated on FIFO channels for allowing an ordered exchange of operations done on document replicas. A replica log contains all operations that have been generated locally or received from other users. Logs are created and updated at user sites. The log structure is defined as:

Definition 1. *Let $\mathbb{P}$ be a set of operations $\{$insert, delete, update, share$\}$ that users can generate; and let $\mathbb{T}$ be a set of event types $\{$write, share, contract$\}$. An event e is defined as a triplet of $\langle$evt $\in \mathbb{T}$, op $\in \mathbb{P}$, attr$\rangle$, in which evt is the event type, op is an operation and attr includes attribute pairs of $\{$attr_name, attr_value$\}$ to present extended information for each event.*

Definition 2. *A document log L is defined as an append-only ordered list of events in the form $[e_1, e_2, \ldots, e_n]$.*

In Listing 1 we give an example of a log containing a single event that has three attributes.

```
<log> <!-- at site D -->
  <event>
   <evt>write</evt>
   <op>insert</op>
   <attr>
      <by>User D</by>
      <content>first lines of document</content>
      <gsn>1</gsn>
   </attr>
  </event>
</log>
```

Listing 1. An example of log with one event in XML format

Users store operations in their logs in an order that is consistent with the generated order. The *share* operation is issued and logged when a user pushes his changes. A *share* operation can be followed in the log by *contract* operations representing usage policies for the shared data.

C. Contracts

In our model, a contract expresses usage policy which one user expects others to respect when they receive and use shared data. Contracts are built on the top of a basic deontic logic [5] with the normative concepts of obligation, permission and forbiddance representing what one ought to, may, or must not do. We use the same notion of contract as in [16].

```
<log> <!-- at site B -->
  <event>
   <evt>write</evt><op>insert</op>
   <attr>
      <by>User D</by>
      <content>first lines of document</content>
      <gsn>1</gsn>
   </attr>
  </event>
  <event>
   <evt>share</evt><op>share</op>
   <attr>
      <by>User D</by><to>User B</to>
      <gsn>2</gsn>
      <rsn>1</rsn>
   </attr>
  </event>
  <event>
   <evt>contract</evt><op>delete</op>
   <attr>
      <by>User D</by><to>User B</to>
      <modal>F</modal>
      <gsn>3</gsn>
      <rsn>2</rsn>
   </attr>
  </event>
  <event>
   <evt>contract</evt><op>share</op>
   <attr>
      <by>User D</by><to>User B</to>
      <modal>P</modal>
      <gsn>4</gsn>
      <rsn>3</rsn>
   </attr>
  </event>
</log>
```

Listing 2. An example of log containing contract events

1) Contract definition: A contract is denoted by a modality followed by an action. Unlike access control policy-based model [24] where an enforcing policy is integrated in the replication protocol, there is no entity in our system that enforces users to follow contracts. Instead, each user performs a self-auditing on collaboration logs to analyze whether users respected the given contracts. Once any misbehavior not conforming to given contracts is detected, the trust level of the user who misbehaved will be decremented.

A contract which is a statement of data usage willingness one user gives to intended receiving user is defined as follows.

Definition 3. *Let a contract primitive be a log-event that has a modality attribute (called modal) equal with P (permission), O (obligation), F (forbiddance) or M (omission). If op is an operation then the contract primitive* c_{op} *based on op is denoted as:* F_{op}, O_{op}, P_{op}, M_{op}. *A contract* C *is defined as a set of one or several contract primitives.*

For example, in Fig. 1, user D gives to user B a contract $\mathsf{C}_{F_{\text{delete}},\,P_{\text{share}}}$ (*forbiddance to delete* and *permission to share*). The two contract primitives are F_{delete} and P_{share}. When a user shares data by means of a push primitive, a contract is logged as events with attributes representing users who sent and received the contract. In Listing 2 we illustrate the representation of the log at site B after user B cloned document *Doc* from user D.

The event attribute *GSN* (*generate sequence number*) of an event is equal to the logical clock [9] at the site where the event was generated. The event attribute *RSN* (*receive sequence number*) of a *share* or *contract* event is equal to the logical clock of the receiving site at the reception of this event. We will discuss how to manage these sequence numbers in the next section.

If c_{op} is a contract primitive then $c_{\neg\text{op}}$, $\neg\, c_{\text{op}}$, $\neg\, c_{\neg\text{op}}$ are contract primitives too. For instance, $c_{O_{op}}$ denotes an obligation to perform *op* and $c_{P_{op}}$ denotes a permission to perform *op*. If we have n contract primitives, we can obtain a contract by merging these contract primitives. For instance, if we have two contract primitives $\mathsf{C}_{P_{\text{op}_1}}$ (permission of performing op_1) and $\mathsf{C}_{O_{\text{op}_2}}$ (obligation of performing op_2), then we can build the contract $\mathsf{C}_{P_{\text{op}_1},\,O_{\text{op}_2}}$. Concerning merging contract primitives to obtain a contract, we define the following axioms:

(A1) $\mathsf{C}\ c_{\neg\text{op}} \rightarrow \neg\ \mathsf{C}\ c_{\text{op}}$

(A2) $\mathsf{C}\ c_1 \wedge (c_1 \rightarrow c_2) \rightarrow \mathsf{C}\ c_2$.

(A3) $\mathsf{C}_{c^1_{\text{op}} > \ldots > c^n_{\text{op}}} \rightarrow \mathsf{C}_{c^1_{\text{op}}}$.

The axiom (A1) means that if C says $c_{\neg\text{op}}$ then it is deducible C does not say c_{op}. The axiom (A2) shows the consequent deducibility. For instance, if we suppose that $O_{\text{update}} \rightarrow P_{\text{insert}}$, then $\mathsf{C}_{O_{\text{update}}} \wedge (O_{\text{update}} \rightarrow P_{\text{insert}}) \rightarrow \mathsf{C}_{P_{\text{insert}}}$. The axiom (A3) rules the merging process. It means that if we have *n* contract primitives $c^1_{\text{op}} > \ldots > c^n_{\text{op}}$ then these contract primitives can be merged and the resulting contract is deducible as $\mathsf{C}_{c^1_{\text{op}}}$. For instance, if $\mathsf{C}_{O_{\text{op}},\,P_{\neg\text{op}}}$ and $O_{\text{op}} > P_{\neg\text{op}}$ then it is deducible to have a contract $\mathsf{C}_{O_{\text{op}}}$. This means that even if $\neg op$ is permitted to be performed with the contract $\mathsf{C}_{P_{\neg\text{op}}}$, there is an obligation to perform *op* if $\mathsf{C}_{P_{\neg\text{op}}}$ and $\mathsf{C}_{O_{\text{op}}}$ are merged as $O_{\text{op}} > P_{\neg\text{op}}$.

2) Contract conflicts: Although each contract is conflict free, conflicts may arise when contracts are merged. There are two cases of conflict that are weak conflict, for instance, a conflict between *permission* and *forbiddance* or between *obligation* and *omission*; and strong conflict when a contract refers to do and not to do the same action, for instance, a conflict between *obligation* and *forbiddance*.

Definition 4. *Two contracts* C_1 *and* C_2 *conflict (denoted as* $\mathsf{C}_1 \lightning \mathsf{C}_2$*) if one primitive contract* $c_i \in \mathsf{C}_1$ *conflicts with another primitive contract* $c_j \in \mathsf{C}_2$. *Primitive contracts are conflicting in the following cases:* $O_{\text{op}} \lightning O_{\neg\text{op}}$, $P_{\text{op}} \lightning F_{\text{op}}$, *and* $O_{\text{op}} \lightning M_{\text{op}}$.

We discuss in what follows how conflicts are resolved. Contracts are interrelated and interdependent and there is no hierarchy between them. Thus, some priorities can be established in terms of specific objectives and they vary depending on the system. One way to select a contract in the case of a conflict is to assign orders to various contracts, and select the one with the highest or lowest order. We describe afterward a method for ordering contracts based on the order of operations associated with them.

Definition 5. *The order of a contract primitive is a value assigned to it that determines the priority of the contract primitive with respect to another contract primitive.*

Depending on the operations, the order of contract primitives are given as follows. Operations are categorized to different groups. Given two operations op_1 and op_2 with the order $\mathrm{op}_1 > \mathrm{op}_2$, then the order of contract primitives is assigned according to the order of operations, $\mathrm{c}_{\mathrm{op}_1} > \mathrm{c}_{\mathrm{op}_2}$. If the contract primitives associated with operations belong to different groups, then we determine a combined order for each, based on the order within group and the order of the group. We discuss below the comparison of two contracts.

Let $\mathbb{P}$ be a set of n operations that could be ordered as $[\mathrm{op}_1, \mathrm{op}_2, \ldots, \mathrm{op}_\mathrm{n}]$ from highest to lowest priority, $\mathbb{S}$ be a set of n-digit ternary numbers from 0 to 3^n and a contract C including contract primitives built over operations of $\mathbb{P}$. A mapping from C to $\mathbb{S}$ has as result $s \in \mathbb{S}$ where: (1) if $\mathrm{O}_{\mathrm{op_i}} \vee \mathrm{F}_{\mathrm{op_i}} \in \mathsf{C}$, $s[i] = 0$; (2) if $\mathrm{M}_{\mathrm{op_i}} \vee \mathrm{P}_{\mathrm{op_i}} \in \mathsf{C}$, $s[i] = 2$; (3) if neither (1) nor (2) are true, $s[i] = 1$. The comparison of two contracts C_1 and C_2 is based on the comparison of s_1 and s_2 and $(\mathsf{C}_1 > \mathsf{C}_2) \Leftrightarrow (s_1 > s_2)$. For instance, given a set $\mathbb{P}$ of two operations (n=2) in the order $[\mathrm{op}_1, \mathrm{op}_2]$, we suppose that we want to compare two contracts $\mathsf{C}_1 = [\mathrm{O}_{\mathrm{op}_1}, \mathrm{P}_{\mathrm{op}_2}]$ and $\mathsf{C}_2 = [\mathrm{O}_{\mathrm{op}_2}]$. The 2-digit ternary numbers $s_1 = 02$ and $s_2 = 10$ are mapped from C_1, C_2 to $\mathbb{S}$. We have $s_2 > s_1$, so $[\mathrm{O}_{\mathrm{op}_2}] > [\mathrm{O}_{\mathrm{op}_1}, \mathrm{P}_{\mathrm{op}_2}]$, hence we have $\mathsf{C}_2 > \mathsf{C}_1$.

3) Contract dominance: A primitive contract c_2 is said to dominate one another c_1 if c_2 is of higher order than c_1 or c_2 overrides c_1 where c_2 overrides c_1 means $(\mathrm{c}_1.\mathrm{op} = \mathrm{c}_2.\mathrm{op}) \wedge$ (c_1`.attr.by` = c_2`.attr.by`) $\wedge$ (c_1`.attr.to` = c_2`.attr.to`) $\wedge$ (c_2 was received after c_1).

IV. Collaborative Process

In this section, we present different aspects of a collaborative process involving our C-PPC model: logging changes, pushing logs containing document modifications and contracts, and merging pairwise logs.

A. Logging changes

Each site maintains a local *clock* to count events (*write*, *share*, and *contract*) received from remote sites. As the changes are made or received, they are added to log in the following manner:

- When a site generates a new event e, it adds e to the end of local log in the order of occurrence and augments its clock. The *clock* value will be assigned to the attribute *GSN* (*generate sequence number*) of event e, i.e. `e.attr.GSN = clock`.
- When a site receives a log from another site, the events from the remote log that are new to the local log are appended at the end of the local log in the same order as in the remote log.
- When a user shares a document with another user, he sends a share event followed by some contracts so-called e which are logged by the receiving user. We denote by e one of these events (share or contract). At the time of reception, the receiver assigns its clock to the attribute *RSN receive sequence number* of e, i.e. `e.attr.RSN = clock`.

We assume that a user is unwilling to disclose to all collaborating users all the sharing events and contracts that he gave to a certain user. Thus *share* events and contracts are not kept in the log of the sender. Moreover, even if a site sends those events to other sites, the receiving sites could refuse integration of remote changes. In this way sending sites would contain share and contract events that have not been accepted by receivers. Therefore, *share* events and contracts are not logged by the sending site.

B. Pushing logs containing contracts

Contracts are given when a user shares a document to another user. The user pushes his log as follows:

- Since a document is shared as a log of operations, therefore to send a contract for document usage control, the contract is attached at the end of the log.
- In sharing, a user specifies a new contract; however, he cannot specify a higher contract than what he currently holds. For instance, if a user u currently holds a contract C on the document d, he only can share d with another user with a contract C' that $\mathsf{C}' \leq \mathsf{C}$.
- A user cannot specify a new contract which conflicts with his current contract. For instance, if a user u has a contract $\mathsf{C}_{\ O_{\mathrm{op}}}$, then he cannot add $\mathrm{O}_{\neg\mathrm{op}}$ or M_{op} to C.
- The contracts a user specifies to two distinguished users might be different. These two users do not not know the contract of the other user as far as they do not collaborate with each other.

During the collaborative process the log of each site grows and the document and contracts are updated each time a user synchronizes with other users.

C. Merging pairwise logs

The collaboration involves logs reconciliation. When a user u receives a remote log L' from a remote user v through anti-entropy propagation, u elects new events from L' to append to his log L.

```
function merging_check(u,v,L,L',CT,CT'):
   IF Trust(u,v) is low
      # v is distrustful
      result = NoMerge-NoBranch
   ELSE:
      IF conflict(ct' in CT', ct in CT)
         result = Branch-Reject
      ELSE:
         result = Merge
      ENDIF
      return result
   ENDIF
```

Listing 3. Pseudo-code for merging check

A site might receive a remote log with conflicting contracts. In the case of conflicts, the user decides either to reject the remote document version or to create a new branch for the remote version or to leave the local version to accept new one. The function to check for conflict before merging is presented in Listing 3. A site neither merges nor creates a new branch if the sender is distrustful. We have the condition to create a new branch if the remote log includes any primitive contract which conflicts with current contract. We consider dominance of contract if a user revokes an old contract and replaces it by a new one. For instance, the old contract F_{share} received by site v from site u can be replaced by a new one P_{share}. Two logs can be merged if no conflict is found.

If the result returned by `merging_check` function is `Merge`, i.e. the merge can be performed, we use anti-entropy updating and perform synchronization by using our proposed merging mechanism. We assume the merging mechanism ensures causality not only between *write* operations, but also between *share* operations and *contracts*. We next discuss in detail how to ensure causality.

An event e is said committed by site u if it is firstly appended to the log of site u after its generation. To determine the total order of events committed by one site, we use the "commit sequence number" `CSN`. In merge function presented in Listing 4, commit sequence number `CSN` is used to track the last event committed by one site.

As we mentioned before, the attributes of event e, `e.attr.GSN` and `e.attr.RSN`, record the values of the clock of its generation and its receipt, respectively. Note that though `GSN` is assigned to events before a log is propagated, `RSN` is assigned to *share* event and *contract* at the receiving site during the synchronization.

The value of `CSN` of an event e committed by site u is computed as follows:

- If e is a *write* operation generated by u, the commit sequence number `CSN` is equal to the value of attribute `e.atrr.GSN`. The site who commited e is extracted from e's attribute `e.attr.by`.
- If e is a *share* operation or a *contract* primitive given by site v to site u and accepted by site u, the commit sequence number `CSN` is equal to the value of attribute `e.attr.RSN`. The site who commits e is extracted from attribute `e.attr.to`.

The data structure of log and the merging mechanism ensuring that new events are added only to the end of log enable a property that if the log of a site u contains an event e committed by v with a commit sequence number `CSN`, then it has contained all the events committed by v prior to CSN. In order to avoid merging events that have been already integrated, we use a local-vector LV whose maximum size is the number of all collaborators to keep the highest commit sequence number `CSN` of each site $v \neq u$ known by u (highest number `CSN` is kept in $LV_u[v]$). This allows a site u to correctly determine that an event from site v should be merged into local log if its `CSN` is higher than the current entry value of `LV` corresponding to its belonging site.

It is possible to replay *write* operations from the log to get the document state. We can use any existing approaches of communicative replicated data type (CRDT) such as Logoot [23] in which concurrent operations can be replayed in any causal order as they are designed to commute.

The complexity of merging mechanism illustrated in Listing 4 is $O(n)$ where n is the size of the remote log L'.

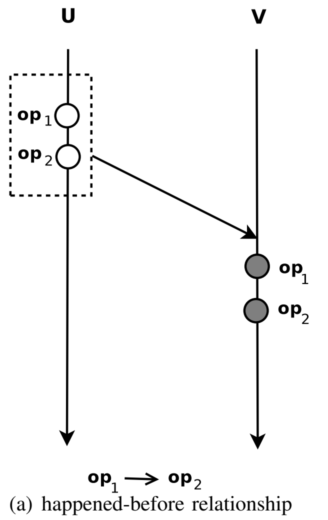
```
function merge(L, L', clock):
   FOR i = 1 to sizeOf(L')
      e := L'[i]
      IF e.evt == ``write'':
         CSN := e.attr.GSN
         site := e.attr.by
      ELSE:
         IF e.attr.RSN == null
            e.attr.RSN = clock
            clock := clock +1
         ENDIF
         site = e.attr.to
         CSN = e.attr.RSN
      ENDIF
      IF CSN > LV[site]
         append e to the end of L
         LV[site] = CSN
      ENDIF
   ENDFOR
```

Listing 4. Pseudo-code for merging local log L with remote log L'

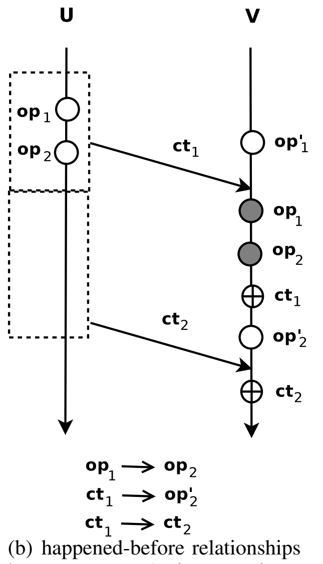
V. Consistency of C-PPC Model

A main issue in maintaining consistency of our C-PPC model is that changes of one site are not broastcasted to all other sites since users' trust levels are different and sites might receive different contracts for the same document state. We discuss the consistency of proposed model based on the well known CCI consistency model [21] which requires the *convergence* of document, the *causality* and *intention preservation*.

Concerning causality preservation, our model deals with two causal relationships: *happened-before* relation and *semantic causality* relation.



(a) happened-before relationship between write operations generated at one site



(b) happened-before relationships between events (write operations op_j and contracts ct_k) of two sites

Fig. 2. Causal relations

1) Happened-before relation "$\rightarrow$": Any two events e_1 and e_2 are in a happened-before relation [9] $e_1 \rightarrow e_2$ if:

(i) for two *write* operations generated at the same site e_1 and e_2, if e_1 was generated before e_2 then $e_1 \rightarrow e_2$. In detail we have $(e_1 \rightarrow e_2) \Leftrightarrow (e_1.\texttt{attr.by} = e_2.\texttt{attr.by})$ and $(e_1.\texttt{attr.GSN} < e_2.\texttt{attr.GSN})$ and $(e_1.\texttt{evt} = e_2.\texttt{evt} = \texttt{write})$ (example in Fig.2.(a))

(ii) for any two events generated by different sites, e_1 generated by site u and e_2 generated by site v, $e_1 \rightarrow e_2$ if e_2 is committed after the arrival of e_1 at site v (example in Fig.2(b))

In our model logs are propagated by using anti-entropy [3] which ensures the causality without using state vectors [12] or causal barriers [17].

2) Semantic causal relation: Two contracts e_1 and e_2 are said to be in a semantic causal relation if e_1 is committed by site u before that e_2 is given to another site. The contracts one site gives to other sites should depend on his current contracts: $(e_1.\texttt{evt} = e_2.\texttt{evt} = \texttt{``contract''})$ and $(e_1.\texttt{attr.to} = e_2.\texttt{attr.by})$ and $(e_1.\texttt{attr.RSN} < e_2.\texttt{attr.GSN})$.

The causality in our C-PPC model is preserved if log is not tampered. We use authenticators for patches of events to detect any attacks to the log; however due to the space limitation, in this paper we omit our solution about the construction and verification of authenticators. Authenticators prevent re-ordering of log events and therefore the causality is preserved. If log was tampered, receiving site might discard it and the trust level of the site that misbehaved is decremented.

Concerning convergence, as our C-PPC model uses CRDT for commutative operations it ensures that in the presence of different contracts received by different sites when the same set of *write* operations was executed at those sites, their copies of the shared document are identical. However the shared document might be in different states on two sites since the shared document is not equally distributed due to the use of contracts and the trust levels of users. And finally concerning the property of intention preservation of the proposed model, it is ensured by causality preservation and CRDT approach [23].

VI. Trust Assessment based on log-auditing

We consider a collaborating system where each user is supposed to respect given contracts. If he does, then the user is *trustful*; otherwise the user is *distrustful* or *suspicious*. There are two ways in which a user cannot be *trustful*: he can either do actions violating a contract or ignore an obligation that needs to be fulfilled. Ideally, if a user misbehaves in either of these ways, other users should detect his misbehavior. A user is considered as *distrustful* if it violates a contract, and a user is considered as *suspicious* if he does not prove that he conforms to an obligation. For instance, a user that receives the obligation *"obligation to insert"* but he never fulfills this obligation is considered *suspicious*. Note that a user u withdraws the suspicious indication on a user v if the user v fulfilled the obligations. Malicious users may try to hide their misbehavior by tampering the log. Briefly, a user u is $malicious$ if he re-orders, inserts or deletes events in the log that consequently affect the auditing result. For instance, u removes some obligations that he does not want to fulfill. The log auditing mechanism should guarantee that no log is tampered. The audit mechanism returns four types of auditing-results: *trustful*, *distrustful*, *suspicious* and *malicious*.

A site can call the auditing protocol at any time. We denote $Trust^L_u(v)$ as the trust value that a user u assigns to a user v. All users are set an initial default trust value, for example their social trusts. A user u updates $Trust^L_u(v)$ for a user v mainly based on the log-auditing result. In order to manage trust levels, we can use an existing decentralized trust model such as [19], [7]. When a user v is assessed as *distrustful* or *suspicious*, his local trust level is recomputed by a user u. In general, a total trust level of a user could be aggregated from log-based trust, reputation and recommendation trust. A trust computation to get the total trust level varies from trust models. The details of our trust model are not presented in this paper. More details about the log auditing and trust assessment mechanisms can be found in [22].

In order to illustrate the trust assessment mechanism, let turn back to our previous example. In the example we presented in Fig. 1 at the beginning of the scenario user C is *trustful* for user B, having the associated trust level equal with 0.3. During the illustrated scenario we have seen that as a result of a log-auditing mechanism user B discovers that user C did not respect some received contracts. User B therefore classifies user C as *distrustful* and decrements his trust level to 0.1. We can see that user B stops afterward the collaboration with user C because of his low trust level.

VII. Evaluation

In this section we present the evaluation of our proposed model by performing some experiments using a peer-to-peer simulator and we discuss some potential limitations.

A. Experiments

We evaluate the feasibility of the proposed model through simulation using PeerSim [13] simulator. We focus first on the ability of detecting misbehaving users; then we estimate the overhead generated by using contracts. We setup the simulation with a network of 200 users where some of them are defined as misbehaving users. Due to the unavailability of real data traces of collaboration including contracts, we generate randomly the data flow of collaboration during the simulation, i.e operations, contracts and users with whom to share. One interaction is defined as the process of sharing a log with the specified contracts, from one user to another one. Since the total number of interactions generated must be pseudo uniformly distributed over all users, we let one user perform sharing with no more than 3 other users at each step. Similarly, the number of operations and contracts generated by one user each time is at most 10 operations and 3 contracts (if we consider only 3 types of actions in our system: insertion, deletion and sharing).

1) Experiment 1 - Misbehavior detection: To evaluate the ability of misbehavior detection, we check first the ability to detect a selected misbehaving user according to the total number of interactions performed by all users. The estimation is performed on the collaborative network with 60 misbehaving users (30% of users are misbehaving users). The auditing process is performed after each synchronization with another user. We select randomly one misbehaving user to be audited and we analyse the percentage of users that can detect him. Fig. 3 shows the results collected after each cycle. We can see that the misbehaving user is detected by a few users at the beginning and then the number of users that detect his misbehaviour increases with the number of interactions.

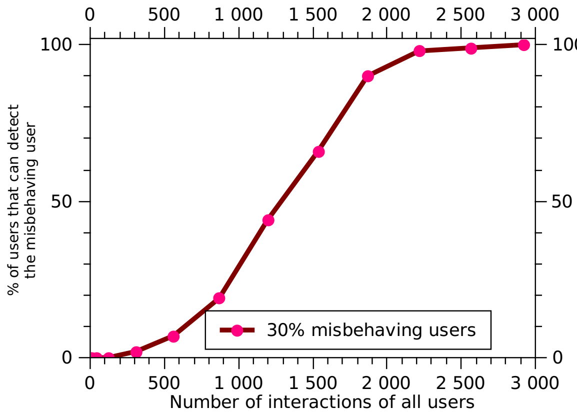


Fig. 3. Ability to detect one selected misbehaving user with respect to the total number of interactions in the collaborative network.

Second, we check the percentage of misbehaving users that can be detected. We select randomly one honest user from the network to observe the percentage of misbehaving users he can detect. Fig. 4 shows the result according to the number of synchronizations done by the selected user with others. We can see from the graph, that up to 20% of misbehaving users are detected after the first four synchronizations (audit done four times), and after the fifth synchronization more than 80% of misbehaving users are detected. We can see a drastic change in the figure between the fourth and the fifth synchronization. That change is due to a synchronization of the selected user with a remote log that contains misbehaviors of most remaining misbehaving users. This can occur in distributed networks of random topology where clusters of collaborating users exist. Once an interaction occurs between two users belonging to such clusters, misbehaving users of the two clusters are discovered. Only about 10% of misbehaving users may require more interactions to be detected. From the results of Fig. 4 we can see that the ability to detect misbehaving users depends also on the topology of collaborative networks. In the future work we will perform more experiments to evaluate how topology would affect the detection.

In order to have a global view about the evolution of the percentage of detected misbehaving users, we compute the average value of detected misbehaving users over all users

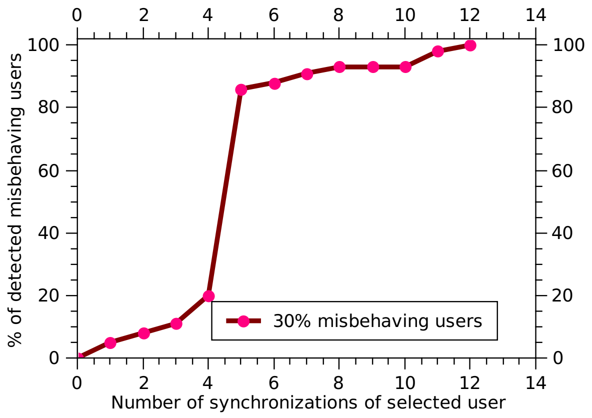


Fig. 4. Percentage of detected misbehaving users with respect to the number of synchronizations done by the selected honest user.

of the collaborative network. Fig. 5 shows, on average, the percentage of misbehaving users that are detected by one user. We perform the experiment in case of a low, medium and high population of misbehaving users in the network (respectively 5%, 30%, 80% of misbehaving users). The results show that the system still functions well in case of a high/low population of misbehaving users.

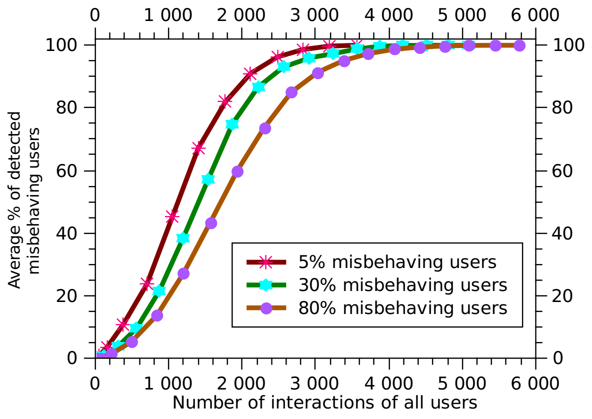


Fig. 5. Average percentage of detected misbehaving users with respect to to the total number of interactions in the collaborative network.

2) Experiment 2 - Overhead estimation: In this experiment we evaluate the time overhead generated by using contracts for the synchronization and auditing mechanisms. We compare two models: with and without contracts. To be able to make the comparison between these two models, we follow the same data flow. In the model without contracts, the synchronization mechanism requires merging logs of operations. In the model with contracts the synchronization mechanism requires merging logs of operations and contracts. Additionally, an auditing mechanism for detection of user misbehaviour has to be applied.

We compute for each model the total time (T) of all the

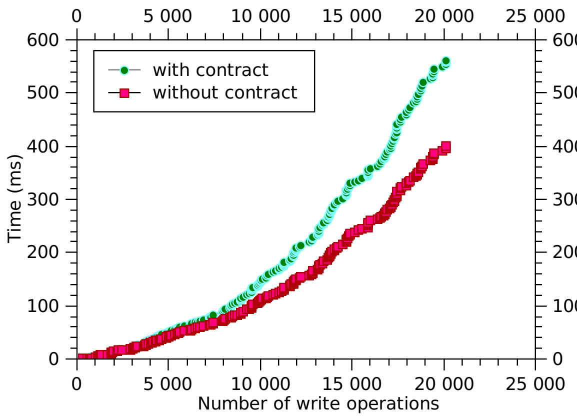


Fig. 6. Synchronization time with growing of number of operations

synchronizations performed by a given user to build the same state of document, i.e. T = $\sum t_i$, where t_i is the time required for the i^{th} synchronization. Fig. 6 shows the result according to the number of write operations in the local log. From the obtained results we can see that the time overhead generated by using contracts is reasonable since the difference between the times computed for the two models increases slowly with the number of operations.

B. Limitations

Our work has some potential limitations. First, contract-based collaboration does not offer a solution for plagiarism and violation of contracts outside of the system. Beyond *write*, *share* operations and contracts that a computer system could log, there are always side channels that can work around the logging. For example, a malicious user could replay write operations from the log to create a new document and then share it and claim himself as being the owner. Or a malicious user could reveal the content of the document outside of the system by using communication means such as email, telephone call and chat, these actions being not logged by the system. These violations can be detected by humans or by using plagiarism techniques, however, this is out of the scope of this paper. The proposed model uses contracts as a means to express data usage restrictions that helps to protect data privacy and to build a trustworthy collaborative environment. A second limitation of our approach is how to deal with the growing size of the log during collaborative process. The log should be ultimately truncated. That requires some additional constraints and consensus of collaborators. At the moment we do not consider log truncation in the proposed model. Third, we have not fully explored a wide range of contracts that can be specified in our editing model. Currently, contracts are based on a basic deontic logic including permission, forbiddance, obligation and omission. They can be combined with operators from temporal logic to express time dimension of contracts, however, we will consider this in our further work.

VIII. Related Work

Distributed version control systems adopt the push-pull-clone model of collaboration, but users are uniformly trusted and there are no contracts specified during collaboration. Wikipedia features an informal contract-based model where contracts are checked by crowd sourcing. Anybody can edit according to rules that are checked a-posteriori by other people. In contract-based models rules have to be explicitly expressed and checked by the system. Contract-based models such as hippocratic databases [1] were mainly applied for centralized systems where contracts can be verified by a central authority. In our approach we applied a contract-based model for a PPC collaboration where there is no central log that can be audited.

Access control mechanisms ensure to give access only to authorized users and this checking is performed before access is allowed. Our contract-based collaboration model gives access first to data without control but with restrictions that are verified a-posteriori. Our C-PCC editing model is closely related to the approach proposed in [24] in terms of ensuring security and privacy in a weakly consistent replication system where users are not uniformly trusted. In [24], access control policy claims are treated as data items. The guards added to replication protocol enforce specified policies at the synchronisation step. A replica must check whether the requested action is allowed by the policy and then decide whether to accept or deny updates. In this approach each replica is a local authority that maintains current policies. This is similar to our approach where we let each user perform self-auditing based on local view of other users actions. However, the approach described in [24] only expresses rights but not obligations that each replica should follow. Moreover, only the author of an item can define the policy associated to it and hence there is no need to resolve conflicts between policies. In our approach we need to deal with policy conflicts as multiple contributors can specify different contracts on the shared document. Moreover, the system uses a state-based replication where each site applies updates to its replica without maintaining a change log rather than an operation-based replication as in our work.

Trust management is an important aspect of the solution that we proposed. The concept of trust in different communities varies according to how it is computed and used. Our work relies on the concept of trust which is based on past user behaviors [14]. With our C-PPC model users first bring social trust into the system. However trust is not immutable and it changes over time. Thus trust should be managed by using a trust model. Various trust models for decentralized systems exist such as NICE model [19], EigenTrust model [7]. A trust model includes three basic components [11]: gathering behavioral information, scoring and ranking peers and rewarding or punishing peers. Most of existing P2P trust models propose mechanisms to update trust values based on direct interactions between peers while we use log auditing to help one user evaluate others either through direct or indirect interactions. Our mechanism for discovering misbehaving users can be coupled with any existing trust model in order to manage user trust values.

Keeping and managing event logs is frequently used for ensuring security and privacy. This approach has been studied in many works. In [2], a log auditing approach is used for detecting misbehavior in collaborative work environments, where a small group of users share a large number of documents and policies. In [8], [18], authors present a logical policy-centric framework for behaviour-based decision making. The framework consists of a formal model of past behaviors of principals which is based on event structures. However, the models presented in [2], [8], [18] require a central authority that has the ability to observe all actions of all users. This assumption is not valid for a purely distributed PPC collaboration. The complexity of our log-auditing mechanism compared to centralized solutions comes from the fact that each user has only a partial overview of the global collaboration and can audit only users with whom he collaborates. Therefore, a user can take decisions only from the information he possesses from the users with whom he collaborates.

IX. Conclusion

We presented a contract extended push-pull-clone model (C-PPC) where users share their private data by specifying some contracts that receivers should follow. Trust values are adapted according to users' past behavior regarding conformance to received contracts. Modifications done by users on the shared data and the contracts given when data is shared are logged in a distributed manner. A mechanism of distributed log-auditing is applied during collaboration and users that did not conform to the required contracts are detected and therefore their trust levels are updated. We implemented the proposed collaboration model with a number of simulations using PeerSim simulator. Experiment results show the feasibility of our model. In future work, we plan to analyse solutions for truncation of logs, and to explore a wider range of contracts that can be specified in our proposed model.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. We also thank Hyun-Gul Roh for his revision and discussion.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic Databases. In *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002*, pages 143–154, Hong Kong, China, August 2002. VLDB Endowment.

[2] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini. Audit-based Compliance Control. *International Journal of Information Security*, 6(2):133–151, March 2007.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenkcr, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth annual ACM Symposium on Principles of Distributed Computing, PODC'87*, pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.

[4] P. Dewan and H. Shen. Controlling Access in Multiuser Interfaces. *ACM Transactions on Computer-Human Interaction*, 5(1):37–62, March 1998.

[5] R. Hilpinen, editor. *New Studies in Deontic Logic: Norms, Actions, and the Foundations of Ethics*. Reidel Publishing Company, 1981.

[6] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, 26:214–260, June 2001.

[7] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the 12th International Conference on World Wide Web, WWW 2003*, pages 640–651, Budapest, Hungary, May 2003. ACM Press.

[8] K. Krukow, M. Nielsen, and V. Sassone. A Logical Framework for History-based Access Control and Reputation Systems. *Journal of Computer Security*, 16(1):63–101, January 2008.

[9] L. Lamport. Times, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] J. Loeliger. Collaborating with Git. *Linux Magazine*, June 2006.

[11] S. Marti and H. Garcia-Molina. Taxonomy of Trust: Categorizing P2P Reputation Systems. *Computer Networks*, 50:472–484, March 2006.

[12] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, October 1989. Elsevier Science Publishers B. V.

[13] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer, P2P'09*, pages 99–100, Seattle, WA, September 2009.

[14] L. Mui, M. Mohtashemi, and A. Halberstadt. A Computational Model of Trust and Reputation. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS 2002*, pages 2431–2439, Waikoloa, Big Island, Hawaii, January 2002. IEEE Computer Society.

[15] B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.

[16] G. J. Pace and G. Schneider. Challenges in the Specification of Full Contracts. In *Proceedings of the 7th International Conference on Integrated Formal Methods, IFM 2009*, volume 5423, pages 292–306, Düseldorf, Germany, February 2009. Springer-Verlag.

[17] R. Prakash, M. Raynal, and M. Singhal. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing*, 41(2):190–204, March 1997.

[18] M. Roger and J. Goubault-Larrecq. Log Auditing through Model-Checking. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations, CSFW 2001*, pages 220–234, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[19] R. Sherwood, S. Lee, and B. Bhattacharjee. Cooperative Peer Groups in NICE. *Computer Networks*, 50(4):523–544, March 2006.

[20] G. Stevens and V. Wulf. A New Dimension in Access Control: Studying Maintenance Engineering Across Organizational Boundaries. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW 2002*, pages 196–205, New Orleans, Louisiana, USA, November 2002. ACM Press.

[21] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.

[22] H. T. T. Truong and C.-L. Ignat. Log Auditing for Trust Assessment in Peer-to-Peer Collaboration. In *Proceedings of the 10th International Symposium on Parallel and Distributed Computing, ISPDC 2011*, Cluj-Napoca, Romania, July 2011. IEEE Computer Society.

[23] S. Weiss, P. Urso, and P. Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, August 2010.

[24] 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 2010*, pages 293–306, Paris, France, 2010. ACM Press.