BYZANTINE FAULT TOLERANT COLLABORATIVE EDITING

Wenbing Zhao and Mamdouh Babi

Department of Electrical and Computer Engineering
Cleveland State University

Keywords: Byzantine fault tolerance, Collaborative editing, distributed algorithms, optimistic replication, operational transformation, replica consistency.

Abstract

In this paper, we describe a lightweight solution for protecting real-time collaborative editing systems against Byzantine faults. We observe that a centralized coordination algorithm not only reduces the complexity of the editing system, it makes easier to harden the system with Byzantine fault tolerance. We perform a comprehensive analysis of the potential threats towards collaborative editing systems and introduce a set of Byzantine fault tolerance mechanisms without requiring any additional redundant resources. If the system has sufficient redundancy, such mechanisms can be used to ensure strong protection against various malicious faults. Even without sufficient redundancy in the system, our mechanisms would still help limit the damages caused by a faulty user.

1 Introduction

Collaborative editing has been an intense research focus in the field of computer supported cooperative work [3, 6, 13]. Collaborative editing refers to the scenario that a number of users work on a shared document concurrently via a network or the Internet. In recent years, we have seen increasing popularity of real-time collaborative editing systems, such as ACE[1], Gobby, MonEdit, Etherpad, and Google Docs. The demand for such systems lies in their potentially huge financial benefits and the increased productivity for organizations and the convenience for their users. For example, collaborative editing systems could be used to implement virtual teams with members who are geographically apart, are working at home, or are traveling on the road.

The primary challenge for collaborative editing systems is to manage consistency between different users *while allowing them to update the shared document concurrently* (pessimistic concurrency control by serializing all operations is possible, however, it is not desirable due to the substantial delay it may cause to users). Essentially, real-time collaborative editing is an optimistic replication problem [11] because each user possesses a replica of the shared document and the user applies its update to the replica immediately before propagating the update to other users. The convergence to consistent state among the replicas is achieved by observing the causal relationship between update operations and by applying *operational transformation* [3] to concurrent (and hence conflicting) update operations. Operational transformation would bring the replicas state to a consistent state while preserving the users' intent [13].

A number of coordination and operational transformation algorithms have been developed to enable concurrent updates to the shared document while achieving eventual consistency of the state among the users [3, 8, 13]. However, to build robust collaborative editing systems for the Internet environment, such algorithms alone are not sufficient because they (inevitably) contain explicit or implicit single-point of failures (more details given in the next section).

The research on collaborative editing systems is predominately on achieving eventual consistency [6, 13]. The limited number of publications on the fault tolerance aspect [9, 10, 12] all assumed the crash-fault model.

In [9, 10], a primary-backup scheme is used to tolerate a single crash fault at the server. It is assumed that a replicated server is used to coordinate all users of an editing session. In contract, our mechanisms can be used to protect both crash faults and malicious faults. Furthermore, we do not require the use of any additional resources. Another focus in [9, 10] is to reduce the recovery time for a failed client by performing periodic logging of the local state at the client. In a way, this is similar to the checkpointing and logging mechanism introduced in our work. Even though our checkpoing and logging mechanism is designed for a different purpose, it is certainly can be used to minimize the recovery time of crashed participants.

In [12], a stateful group communication system is proposed as the foundation to build fault tolerant groupware applications. The group communication system could alleviate the burden of managing shared document and membership changes at the application layer. Similar to [9, 10], the system can only tolerate crash faults instead of malicious faults. Furthermore, a big concern for this approach is that users in a collaborative editing application running on top of the stateful group communication system would be tightly coupled together by a proprietary communication protocol, and thus, it would be difficult to deploy such applications over the Internet.

We argue that the crash-fault model is not adequate for collaborative editing systems operating in the untrusted Internet environment because we may encounter corrupt users

[1] ACE is the name of the editor, not an acronym.

and computer systems running the editing software might become compromised, both endangering the integrity of the shared document. To the best of our knowledge, our work is the first that aims to achieve Byzantine fault tolerance for real-time collaborative editing systems.

In this paper, we conduct a threat analysis of real-time collaborative editing systems and describe a set of Byzantine fault tolerant mechanisms to mitigate such threats. Here Byzantine fault refers to an arbitrary fault including both crash fault and malicious fault caused by a corrupt user or a compromised system [5]. Despite the fact that there have been numerous Byzantine fault tolerance (BFT) algorithms developed for general-purpose distributed systems [1, 4], they cannot be used to build a robust real-time collaborative editing system in a straightforward manner due to the following reasons:

- Such BFT algorithms concern only the consistency among the replicas and not the validity of the updates to the system state. If a Byzantine faulty user submitted a malicious update to the system, a BFT algorithm only ensures that it is applied to all replicas. It is apparent that this would compromise the integrity of the shared document.
- Such algorithms may block the system for unbounded amount of time in some failure scenarios due to their intrinsic pessimistic nature, *i.e.*, they favour safety over liveness of the system. This is not acceptable for real-time collaborative editing.
- All BFT algorithms require the use of sufficient number of replicas to reach a consensus (often 3f+1 replicas to tolerate up to f faulty replicas). In a collaborative editing system, we must allow as few as two users to collaborate.

This work is along the line of our previous work on building application-aware Byzantine fault tolerance solutions [2, 14]. By exploiting application semantics, much more optimal mechanisms can be developed to protect the applications from malicious faults. This paper makes the following contributions:

- We make a case to favour the use of centralized coordination algorithms for real-time collaborative editing systems.
- We perform a comprehensive threat analysis on collaborative editing systems.
- We present a set of lightweight BFT mechanisms that can be used to protect such editing systems from malicious faults without resorting to additional redundant resources.

2 Background

A real-time collaborative editing system often involves two or more users. Typically, one user would initiate the collaboration by creating a shared document and advertise the intent to other users, if they are present. If another user is interested, it would join the collaboration and obtain a copy of the document (*i.e.*, the user will possess a replica of the shared document). More users could join the collaboration in the same fashion. A user could also opt to leave the collaboration as he/she wishes. For simplicity, we model the shared document as a linear string of characters [6]. Furthermore, we often do not distinguish the user from the software program running at the user's site if it is clear from the context.

When a user introduces an update to the shared document, the following steps would ensue behind the scene (*i.e.*, they are carried out by the collaborative editing system) [11]:

- *Update submission*. When the user inserted or deleted some text via a graphical user interface, the operation is captured by the system and assigned a logical timestamp.
- *Local execution*. The operation is applied at the local copy of the shared document (i.e., the replica at the user's site) immediately.
- *Update propagation*. The operation is encapsulated in a message, together with the assigned logical timestamp (vector clock [7] is often used to assign the timestamp), and is sent to other users.
- *Scheduling of remote operations*. A user may receive several operations submitted by other users concurrently. The operations are examined based on the piggybacked logical timestamps for possible causal relationship between them. The causality is always respected when applying a remote operation at a user. The timestamp also helps to identify if the user is missing any remote operations. Any out-of-ordered operations are queued until the missing operations are received and applied.
- *Conflict detection and reconciliation*. If two operations are concurrent, *i.e.*, neither happens before the other, they are deemed as conflicting with each other. In collaborative editing systems, conflicting operations are reconciled by applying operational transformation [3].

2.1 Correctness criteria

A correct real-time collaborative editing system should ensure the following three correctness criteria [13]:

- *Causality preservation*. For any causally related pair of update operations O_i and O_j, if O_i happens before O_j, then O_i is executed before O_j at all users.
- *State convergence*. When all users stop submitting further updates (*i.e.*, when the system is quiescent), the replicas of the shared document at different users will become the same.
- *Intention preservation*. This requirement is specifically related to conflict reconciliation using operational transformation. When an operation is applied at the submitting site, the user's intent of the operation is automatically preserved. However, when the operation is propagated to other users and if it has to be transformed due to a conflict with another operation (submitted concurrently by

another user). The transformation must preserve the user's intent.

2.2 Tracking causality using vector clocks

For a system that consists of N users collaborating on a shared document, each user maintains a vector clock, *VC*, in the form of an N-element array [7]. For convenience, we refer to the users in the system in terms of their indices, from 0 to N-1. For user *i*, the corresponding element in its vector clock, $VC_i[i]$, represents the number of updates submitted locally at user *i*. The user learns the values for other elements from the timestamps piggybacked with the messages sent by other users.

The rules for using the vector clock are defined as follows:

- On submission of a new operation *O* at user *i*, the operation is assigned the current VC_i value as the timestamp of the operation, *i.e.*, $O.vc = VC_i$.
  - When propagating the operation to other users, the assigned timestamp is piggybacked with the operation.
  - Subsequently, the element *i* of the vector clock at user *i* is incremented by 1, *i.e.*, $VC_i[i] = VC_i[i] + 1$.
- On receiving an operation *O* at user *j*, user *j* updates its vector clock in the following way:
  - For each element $k \neq j$ in the vector clock, $VC_j[k] = \max(VC_j[k], O.vc[k])$

Note that on receiving an operation from user *i*, user *j* might advance its vector clock at an element *k* other than *i* if user *i* receives an operation ahead of *j*. User *j* might want to request a retransmission for that operation. If the communication channel between *i* and *j* does not ensure the first-in-first-out (FIFO) property, user *j* might receive a previously missing operation after an out-of-ordered operation from user *i*, in which case, the vector clock is not advanced.

A user determines if an operation O_i happens before another operation O_j by comparing their vector clock timestamps. O_i happens before O_j if $O_j.vc > O_i.vc$, *i.e.*, for any $k \in \{0...N\}$, $O_j.vc[k] \leq O_i.vc[k]$. If neither $O_j.vc > O_i.vc$, nor $O_j.vc < O_i.vc$, then, the two operations are concurrent and a conflict is detected.

2.3 Update propagation

The algorithms employed by real-time collaborative editing systems can be roughly divided into two categories in terms of update propagation: (1) fully distributed, and (2) centralized.

In fully distributed algorithms, each user is responsible to multicast its update operation to all other users. In centralized algorithms, the collaboration initiator or a dedicated server is designated as the central server and all other users would act as the clients, *i.e.*, the system is essentially organized into a star topology with the server at the centre. A client first sends its update operation to the server, the server then multicasts it to other users, possibly after transforming the operation.

Not surprisingly, centralized algorithms are much simpler and easier to implement because the problem is reduced to a client-server two-party issue [8]. The obvious downside of centralized algorithms is that the server constitutes a single-point of failure of the system, *i.e.*, the system as a whole would fail if the server fails.

Are fully distributed algorithms more robust? The answer is that they are more robust to only certain extent. While the fully distributed algorithms can tolerate a single fault (*i.e.*, when one user fails, no other user would fail until the failed user recovers) if users log all outgoing and incoming operations. If two or more users might fail concurrently, it might prevent the entire system from making progress. Consider the example illustrated in Figure 1.

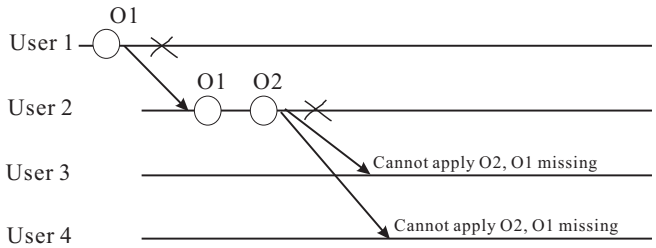

Figure 1. A deadlock scenario due to double faults if a fully distributed algorithm is used.

As shown in Figure 1, user 1 submitted an update locally and then started sending the update operation to other users. Unfortunately, right after user 1 sent its update operation to user 2, site 1 (where user 1 operates) crashed. User 2 then applied the update operation from user 1, and performed a further update to the shared document. Subsequently, user 2 broadcast its update operation to all other users. Unfortunately, before any user realized that user 1's update operation is missing and asked user 2 for retransmission, site 2 also crashed. Because no user could apply user 2's update before receiving user 1's update, and there is no hope for any user to receive the missing update from user 1, hence, no progress can be made from this point on until user 1 or user 2 recovers.

2.4 Operational transformation

Operational transformation is introduced to reconcile conflicting operations [3]. The objective of operational transformation is to achieve state convergence and preserve the user's intent. An example scenario is illustrated in Figure 2 below.

As shown in Figure 2, without proper conflict resolution, the state at different users would diverge. Operational transformation can be used to transform concurrent operations to ensure a converged state at both users. Given two concurrent operations O_1 and O_2, and the transformed operations O_1' and O_2', applying O_1 followed by O_2' would result in the same state when O_2 is followed by O_1'.

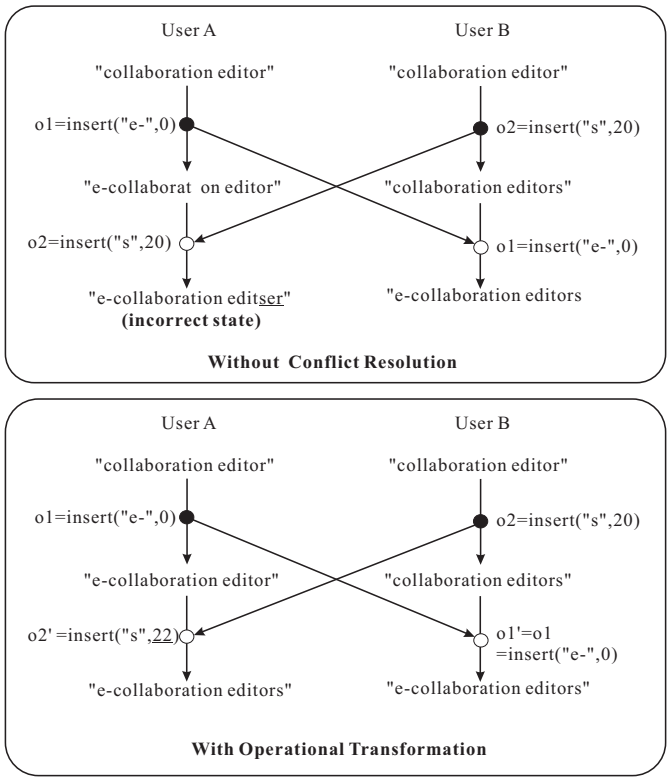

Figure 2. An example scenario with and without operational transformation.

2.5 The ACE collaborative editor

The ACE real-time collaborative text editor is available as an open source project (http://sourceforge.net/projects/ace/). The editor is written in in Java and implements the Jupiter algorithm [8]. Unlike many other open source projects, the ACE editor is comprehensively documented by its developers, which is one of the primary reasons why we choose ACE as the foundation to build a Byzantine fault tolerant collaborative editing system.

The Jupiter algorithm [8] is a centralized algorithm for user coordination and operational transformation. As we will discuss in later sections, the centralized design not only makes user coordination and operational transformation less complicated and less error prone, it also fits the Byzantine fault tolerance design goal very well.

The architecture of the ACE editor is shown in Figure 3. The session server component is created when a user decides to publish a document. The user becomes the *publisher* for the document subsequently. The session server manages the interactions between the publisher and the participants. The session server bears the following responsibilities:

- When a new user requests to join the session, assuming the user is authenticated, the server adds the user to its participant list, and transfers the current document to the new user. Furthermore, the server also notifies existing participants about the joining of the new participant.
- When a user requests to leave the current session, the server removes the user from its participant list and garbage-collect the corresponding data structures. Furthermore, the server also notifies the remaining participants about the leaving of the participant.
- When a participant misbehaves, the publisher would request the session server to kick the participant out of the session and put it in its blacklist. The server then notifies both the participant that is kicked out of the session and all other participants about the membership change.
- The publisher may invite a participant via the session server. When the participant accepts the invitation, all existing participants will be informed about the membership change.
- When the session server receives a message containing an update operation from a participant, it performs operational transformation if it is necessary and applies the update to the shared document. Subsequently, the server forwards (the transformed) operation to all other users. Note that the session server handles all messages sequentially.
- The handling of an update operation from the publisher is rather similar to that from a participant. The only difference is that the session server receives the operation from the publisher via a function call instead of from the network.

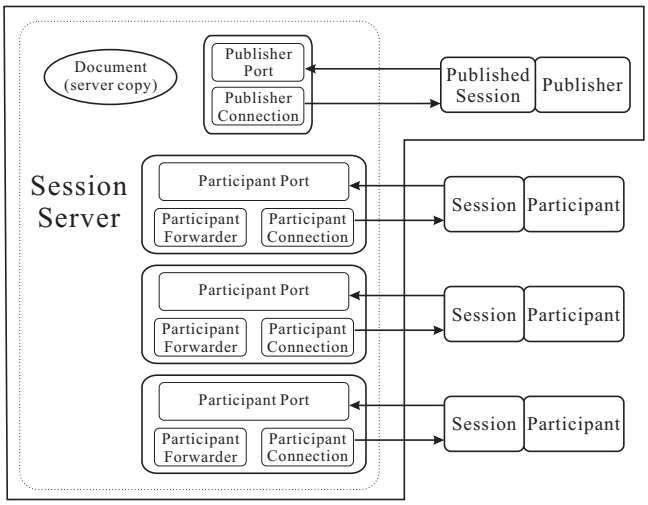

Figure 3. The architecture of the ACE editor with 1 publisher and 3 participants.

We should note that the session server does not necessarily have to be co-located with the publisher. The session server can very well be running as a separate process to provide session services for publishers and participants of shared documents.

The centralized Jupiter algorithm uses 2-way synchronization between the session server and each participant. This design dramatically reduces the complexity of the system. Instead of maintaining a full vector clock for the entire system at each

participant, a participant only keeps track of a vector clock of 2 elements, one for local updates, and the other for remote updates coming from the session server. Essentially, to a participant, all remote updates come from the session server, even if they are originated at other participants.

At the session server, a set of data structure (Participant Port, Participant Forwarder, and Participant Connection, together with an Algorithm object) is allocated to handle the interaction with each participant. Within each set, an independent 2-element vector clock is maintained. The semantics of the vector clock is rather similar to that maintained at the participant. The session server also keeps a server copy of the shared document.

3 Threat Analysis

In this section, we analyse potential threats that could compromise the integrity of a collaborative editing system using a centralized coordination algorithm similar to ACE. We only consider insider threats, including the threats from the publisher or participants of the editing system.

3.1 Threats from a faulty participant

Threat PA1 (malicious updates): The most serious threat from a faulty participant is a malicious update to the shared document, such as introducing faulty texts or deleting texts that should not be deleted. This kind of threats obviously would compromise the integrity of the shared document. To control such threats, the first step is to detect malicious updates. However, the detection of malicious updates can only be done by the publisher or participants because whether or not the shared document has been maliciously altered is application-specific. Once the fault is detected, the effect of the fault can be cancelled by using the undo facility.

Threat PA2 (denial of service attack on the publisher): By design of the collaborative editing system, when a new participant joins an editing session, the publisher (via the session server) would be required to send the current shared document to the participant. A faulty participant could repeatedly join and leave an editing session aiming to launch a form of denial of service attack because the behaviour increases the load on the publisher (the session server to be more specific).

The centralized algorithm has the side benefit that the participants are insulated from each other, that is, a faulty participant cannot directly affect other participants except via threat PA1. If a distributed algorithm is used, a faulty participant could instil another threat to the system by providing wrong information regarding its knowledge about what happened at other participants in the timestamps piggybacked in the messages sent by this faulty participant.

Another problem threat is that a faulty user may refuse the invitation from the publisher. We do not consider this a serious threat because the publisher could invite other participants instead.

3.2 Threats from a faulty publisher

A centralized system puts much more responsibility to the publisher (and the session server) than a system that follows a distributed algorithm. Hence, a faulty publisher can cause much more damages than that in a system using a fully distributed algorithm.

Threat PU1 (malicious updates): A faulty publisher may introduce faulty updates to the shared document, just like any other faulty participant in threat PA1.

Threat PU2 (out-of-service attack): A fault publisher may launch an out-of-service attack on the participants:

- By refusing to accept a join request from a legitimate participant.
- By refusing to relay the updates submitted by one participant to other participants.
- By kicking out a legitimate participant from an editing session.

Threat PU3 (creating artificial partitions): When a participant joins, a faulty publisher may selectively pass on the membership change to a portion of participants and relay the updates submitted by the new participant to them. In effect, a faulty publisher could cause participants to form different membership views and work on completely different versions of the shared document.

Threat PU4 (inconsistent updates): A fault publisher may selectively relay an update submitted by a participant to a subset of the participants. This would cause participants to have different versions of the shared document.

4 Byzantine fault tolerant collaborative editing

In this section, we elaborate Byzantine fault tolerance (BFT) mechanisms that maximize the protection of a collaborative editing system *without resorting to the use of additional redundant resources* (such as a separate Byzantine fault tolerance service cluster). The rational for this design decision is that the publisher and participants are already holding redundant copies of the shared document and performing all the update operations, *i.e.*, there often exist sufficient redundant processes to reach an agreement. Furthermore, requiring additional resources complicates the system maintenance and incurs extra cost for the resources (such as software licenses).

Hence, the mechanisms we introduce here are lightweight, convenient to use, and could be desirable for many practical systems. As a trade-off, not all threats can be mitigated by our mechanisms. In particular, threat PU2 (out-of-service attack) cannot be addressed by our BFT mechanisms. If a user is repeatedly denied service by the publisher, the system administrator will be alerted to resolve the issue. Similarly, threat PU3 (creating artificial partitions) cannot be address by our mechanisms.

Threat PA2 (denial of service attack on the publisher) can be controlled relatively trivially by implementing heuristic

decisions at the publisher. If within a short period of time, a user repeatedly joins and leaves, the user is blacklisted and banned from joining in the future.

The BFT mechanisms introduced in this section are geared primarily towards the control of threat PU4 (inconsistent updates) and to some degree the control of threat PA1 and PU1 (malicious updates).

To tolerate up to f faulty users (publisher or participants), we require the availability of 3f+1 total users. This large total number of users is necessary to achieve Byzantine agreement among the non-faulty users. Of course, the system might encounter failures before sufficient number of participants join the system, or the system might simply do not have enough users. We first introduce Byzantine fault tolerance mechanisms that can be applied when the above condition is met. Then we discuss the reduced protection that can be achieved when the condition is not met.

4.1 System model

We assume that the collaborative editing system is similar to ACE where a centralized algorithm is used to coordinate the participants of a shared document. If the system is deployed in a local area network, users can learn the existing published documents and the participants of each published document automatically via the built-in discovery mechanism (such as Bonjour). If the system is deployed over the Internet, on the other hand, either the publisher posts its contact information (IP address, port, public key, etc.) so that others may request to join an editing session, or the publisher explicitly invites perspective user to join the session. Therefore, we do not assume that the users know each other's contact information unless they have joined the same editing session.

We assume that each user has a public-private key pair. The public key is known to users of the same session, while the private key is kept secret to its owner. Furthermore, all messages exchanged in the system are digitally signed to ensure accountability and to prevent spoofing. We assume that the adversaries have limited computing power so that they cannot break the digital signatures created by non-faulty users.

To clarify, in the description of our BFT mechanisms, when we use the term "user", we mean that the description is applicable to both the publisher and the participant. We use the term "publisher" or "participant" when we want to describe actions/rules specific to the publisher or the participant.

4.2 BFT mechanisms with the presence of sufficient redundancy in the system

In line with the optimistic replication nature of collaborative editing systems, the BFT mechanisms must be optimistic. It would be unacceptable if we require all participants to reach a Byzantine agreement on each message sent by the publisher before they accept the message. Hence, we follow the "trust, but verify" principle when designing the BFT mechanisms. The main strategies used in our BFT mechanisms include:

- *Periodic synchronization*. Periodically, users share with each other their state to see if they are consistent. If not, the publisher is suspected and the inconsistency is resolved by a state transfer or the election of a new publisher.
- *Checkpointing*. At each synchronization point, the shared document is checkpointed as a stable version for possible future recovery.
- *Logging of operations*. All local and remote operations are logged for the purpose of recovery.

Periodic synchronization. To synchronization state, a user would follow the steps illustrated in Figure 4. A critical first step in periodic synchronization is to determine a global synchronization point *without inter-user communication*. We exploit the following property of vector clocks:

- The state of different users would be the same if they have the same vector clock value.

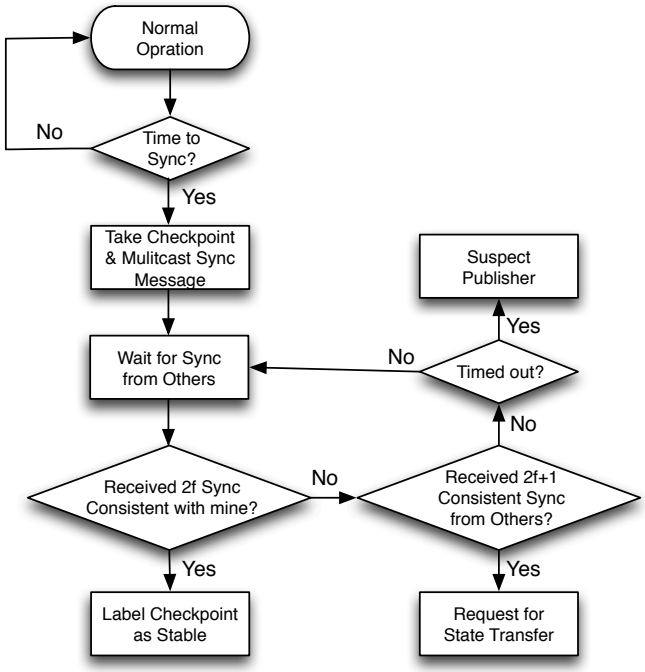

Figure 4. Main steps of periodic synchronization.

We set a user-configurable parameter, *n*, as the interval for synchronization. The rules to determine the synchronization point at each user is given below:

- When a user receives an operation from the session server, if the sum of the local and remote timestamps is equal to *n* or multiple of *n*, it would initiate a new round of synchronization *after* applying the operation assuming that the user has not submitted local operations concurrently. Otherwise, the concurrent local operations are first undone and then the remote operation is applied to ensure a consistent state with the remote user.

- The user that submitted the operation which triggers the round of synchronization at other users would not know it is time to start the round of synchronization because it cannot use the sum of its own vector clock value. Doing so would risk inconsistent state due to concurrent operations at other participants. In this case, the session server must inform the user that it is time to start a new round of synchronization. To prevent a faulty publisher (which co-locates with the session server) from indefinitely delaying the synchronization for this user, a participant starts a timer whenever submitting a candidate operation. If the publisher is not faulty, the user should either receive a synchronization notification, or a remote operation before the timer expires. If it does not, it suspects the publisher.

When a user initiates a round of synchronization, it takes a checkpoint of the current version of the shared document, applies a secure hash on it (such as SHA-2), and multicasts the hash value together with the vector clock value (referred to as a sync message) at the synchronization point to all other users.

When a user collects 2f sync messages from other users that are consistent with its own for the same round of synchronization, it is happy with the current publisher. The user then labels its latest checkpoint as stable. It may proceed to removing all logged operations and the previous stable checkpoint if the garbage collection policy allows.

If a user could collect 2f+1 consistent sync messages from other users, but they are *not* consistent with its own for the same round of synchronization, it suspects the primary, but proceeds to requesting a state transfer from other participants. When it receives the version of the shared document, it replaces its own using the received one and moves forward. The reason why it has to move on is that 2f+1 other participants are happy with the current publisher and hence it does not have enough votes to overturn the current publisher.

Checkpointing and logging. For each round of synchronization, a stable checkpoint of the shared document will be produced at each user. All incoming and outgoing update operations are logged.

If the editing session is relatively short, all checkpoints and logged operations can be preserved. They would be useful for recovery when threats such as PA1 or PU1 (malicious updates) are detected. Quite often, there may be a significant delay between when a user starts to maliciously update the document and when the fault is detected. The logs can be useful for forensic analysis to determine the starting point of the malicious updates. Once the starting point is detected, the document is reversed back to the most recent stable checkpoint of the document prior to the starting point, and the logged operations *excluding those faulty updates* since that stable checkpoint are used to undo the damages done by the malicious user.

If the editing session is long and the physical resources are limited, only the most recent one or several checkpoints and the operations logged since are preserved on the production of a new stable checkpoint. As a trade-off, this garbage collection policy may limit the degree of recovery should threats such as PA1 or PU1 (malicious updates) are detected.

Election of a new publisher. If a participant could not collect 2f+1 consistent synchronization messages from distinct users within a reasonable amount of time (determined by the view change timeout), it suspects the publisher and attempt to elect a new publisher. As shown in Figure 5, a participant would multicast a view-change message naming the oldest participant in its membership as the new publisher.

To speedup the view change process, a non-faulty participant would join the election even if it has not timed out the current publisher, if it has received f+1 view-change messages from other participants.

On receiving 2f view-change messages from other participants, the new publisher would announce the acceptance of the new view and becomes the new publisher. If TCP is used as the transport protocol among all users, this step can often be omitted because it is highly likely this participant would also receive sufficient number of view-change messages when other participants have received 2f+1 view-change messages.

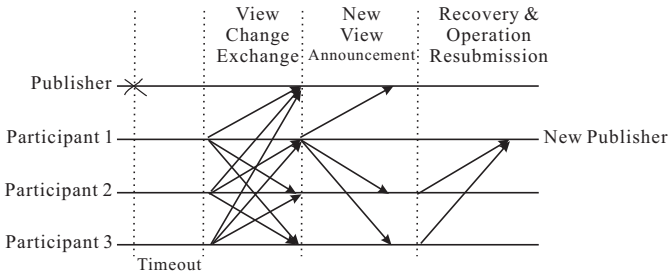

Figure 5. Steps in electing a new publisher.

On receiving the new-view announcement from the new publisher, or as soon as receiving 2f+1 view change messages naming the same participant as the new publisher, a participant reverts its state to the last stable checkpoint and sends its log since the last stable checkpoint to the new publisher. The new publisher also reverts its state to the last stable checkpoint and reprocesses all non-duplicate operations after excluding the operations from the previous publisher since the last stable checkpoint and merging all the submitted operations from other users, prior to handling new operations.

4.3 BFT mechanisms without sufficient redundancy in the system

Without sufficient redundancy in the system, a Byzantine agreement cannot be achieved in the presence of faulty users. The best we could do is to minimize the damages done by a malicious user with the following mechanisms:

- Logging all operations. The log will be useful for forensic analysis and recovery.

- Periodic synchronization of state. Even though no stable checkpoint of the shared document can be guaranteed in the presence of faulty users, non-faulty users could try to resolve any discrepancies by exchanging logged operations when state divergence is detected. This mechanism would be useful to control threat PU4. If inconsistency still persists, the system administrator is alerted.

For long editing sessions, continuous logging might saturate the available storage space and make any attempt to reconcile state inconsistencies by sifting through all the logged operations impractical. This situation can be alleviated by doing garbage collection during fault-free periods once users produce stable checkpoints. In the absence of faulty users, all sync messages would be consistent with each other. Hence, users could deduce that a checkpoint is stable once it knows that all other users have collected consistent sync messages from every other user.

5 Conclusion and future work

The work presented in this paper is focused on real-time collaborative systems that use centralized coordination algorithms. We carried out a comprehensive analysis of the potential threats towards such systems, and introduced a lightweight Byzantine fault tolerance solution for such systems without requiring any additional redundant resources. If the system has sufficient redundancy, our BFT mechanisms can be used to ensure strong protection against various threats. Even if the system does not have sufficient redundancy, our mechanisms would still help limit the damages caused by a faulty publisher.

In future work, we plan to implement the BFT mechanisms described in this paper and incorporate them in the ACE real-time collaborative editing system. We also plan to develop rigorous correctness properties and the corresponding proof of correctness of our BFT mechanisms. Furthermore, we are going to investigate BFT solutions for real-time collaborative editing systems using fully distributed coordination algorithms.

References

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

[2] H. Chai, H. Zhang, W. Zhao, P. M. Melliar-Smith, and L. E. Moser. Toward trustworthy coordination for web service business activities. *IEEE Transactions on Services Computing (to appear)*.

[3] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pp. 399–407, New York, NY, USA (1989)

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of 21st ACM Symposium on Operating Systems Principles* (2007).

[5] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4, pp. 382–401 (1982).

[6] D. Li and R. Li. An admissibility-based operational transformation framework for collaborative editing systems. *Comput. Supported Coop. Work*, 19(1), pp. 1–43 (2010).

[7] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pp. 216–226. Elsevier Science Publishers B.V. (North-Holland) (1989).

[8] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM Symposium on User interface and software technology*, pp. 111–120, New York, NY, USA (1995).

[9] X. Qin and C. Sun. Efficient recovery algorithm in real-time and fault-tolerant collaborative editing systems. In *ACM Workshop on Collaborative Editing Systems*, Philadelphia, Pennsylvania, USA (2000).

[10] X. Qin and C. Sun. Recovery support for internet-based real-time collaborative editing systems. In *Proceedings of the International Conference on Computer Networks and Mobile Computing*, pp. 181, Washington, DC, USA (2001).

[11] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), pp. 42–81 (2005).

[12] H. S. Shim and A. Prakash. Tolerating client and communication failures in distributed groupware systems. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pp. 221, Washington, DC, USA (1998).

[13] 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 Trans. Comput.-Hum. Interact.*, 5(1), pp. 63–108 (1998).

[14] H. Zhang, H. Chai, W. Zhao, P. M. Melliar-Smith, and L. E. Moser. Trustworthy coordination for web service atomic transactions. *IEEE Transactions on Parallel and Distributed Systems*, 23, pp.1551–1565 (2012).