computers

MDPI

Article

Towards Trustworthy Collaborative Editing

Mamdouh Babi [†] and Wenbing Zhao [*,†]

Department of Electrical Engineering and Computer Science, Cleveland State University, 2121 Euclid Ave, Cleveland, OH 44115, USA; mobabi@yahoo.com
* Correspondence: wenbing@ieee.org; Tel.: +1-216-523-7480
† These authors contributed equally to this work.

Academic Editor: Kartik Gopalan
Received: 14 December 2016; Accepted: 28 March 2017; Published: 30 March 2017

Abstract: Real-time collaborative editing applications are drastically different from typical client–server applications in that every participant has a copy of the shared document. In this type of environment, each participant acts as both a client and a server replica. In this article, we elaborate on how to adapt Byzantine fault tolerance (BFT) mechanisms to enhance the trustworthiness of such applications. It is apparent that traditional BFT algorithms cannot be used directly because it would dictate that all updates submitted by participants be applied sequentially, which would defeat the purpose of collaborative editing. The goal of this study is to design and implement an efficient BFT solution by exploiting the application semantics and by doing a threat analysis of these types of applications. Our solution can be considered as a form of optimistic BFT in that local states maintained by each participant may diverge temporarily. The states of the participants are made consistent with each other by a periodic synchronization mechanism.

Keywords: Byzantine fault tolerant; collaborative editing; Byzantine agreement; operational transformation

1. Introduction

By collaborative editing, we mean that two or more participants work on a shared document concurrently via a network. Collaborative editing has been an active research area since the 1980s [1–3]. Collaborative editing systems are desirable because they help increase the productivity for organizations and increase the convenience for employees. The increased popularity of real-time collaborative editing systems in recent years has inevitably elevated the trustworthiness expectation from their users. Considering the untrusted nature of the Internet, there is a strong need to make such applications more robust against various malicious attacks. Byzantine fault tolerance (BFT) appears to be an excellent means to achieve the goal [4]. However, BFT algorithms are designed for client–server applications and they typically require all requests to be executed sequentially to ensure replica consistency. Hence, these algorithms cannot be directly employed for collaborative editing systems.

In designing BFT mechanisms for real-time collaborative editing systems, we must consider the following aspects:

- Replication: With replication, each participant has a copy of the shared document. In this configuration, each participant acts as both a client and a server replica because anyone could make changes to the shared document. Changes made would be propagated to all other participants. In essence, a collaborative editing application is a form of fault tolerance system with active replication [5] and every participant constitutes as a replica. On the other hand, a client–server application configuration by itself does not have built-in replication feature. If fault tolerance is necessary, the server must be explicitly replicated in client–server applications.

The intrinsic replication design in collaborative editing systems makes it very attractive to employ replication-based fault tolerance solutions to increase the trustworthiness. Since the redundancy has already been built into the application, both the hardware cost (i.e., no need to purchase more computers) and runtime overhead are reduced (i.e., the application itself must have incorporated some replica coordination mechanisms).

- Concurrency: Real-time collaborative editing systems are designed to allow concurrent updates to a shared document [6]. Therefore, it is not acceptable to impose any sequential order on the updates to the shared document among all participants because this would be completely against the design purpose of the collaborative editing applications. For typical client–server applications, however, losing concurrency constitutes only a performance issue. For collaborative editing, we will adopt the optimistic replication strategy [7], which means that the states of the replicas could diverge temporarily and it is inevitable for us to use the eventual replica consistency model.
- Role: In collaborative editing applications, each participant acts both as a client and a server. As the client, it may introduce state changes to the shared document. As a server, it will receive and incorporate changes made by other participants. In typical client–server applications, a client only issues requests to the server and anticipates the corresponding replies from the server, and the server passively waits for requests issued by clients and processes the requests and generates replies. Basically, the server provides a function to serve its clients. The server state will not change unless it has processed a request. With the adoption of the optimistic replication strategy, the dual-role of each participant in collaborative editing is no longer an issue.
- Membership: In general, a real-time collaborative editing application only allows a finite set of participants to modify a shared document. For a user to participate, he/she would have to register with the application prior to being granted the privilege to change the shared document. However, client–server applications typically are designed to serve many clients concurrently. Hence, we normally do not need to worry about the scalability issues for collaborative editing applications.

The goal of this study is to design and implement an efficient BFT solution by exploiting the application semantics and by doing a threat analysis of these types of applications. In our solution, each participant is free to introduce changes to the shared document and the propagation of the changes to other participants is done as usual. Our mechanisms ensure eventual replica consistency by synchronizing the states of the replicas periodically.

The remaining of the article is organized as follows. Section 2 presents background information and related work. Section 3 elaborates on the methods used in this research, which includes a threat analysis, system model, membership, and state synchronization. Section 4 provides experimental results. Section 5 concludes this article.

2. Background

2.1. Collaborative Editing

Real-time collaborative editors facilitate multiple participants to work on a shared document concurrently [8,9]. Typically, one user initiates the collaboration by creating a document and making it available (i.e., publish it) to other users. In order to achieve consistency and good response, the shared document and related data (such as membership) are replicated on all sites allowing any participant to edit the document at any time. When two participants generate concurrent operations (O1, O2), the system has to ensure convergence to the best possible state.

Most common mechanisms for collaborative editing applications are based on Operational Transformation (OT). OT was introduced to reconcile conflicting operations [1,10,11]. OT algorithms should ensure causality, convergence and intention preservation for real-time collaborative editing systems. As shown in Figure 1, an operation O1 generated at site S1 by participant 1 is executed locally immediately and broadcasted to all other sites to be re-executed (e.g., at site S2 for participant 2). Without a proper mechanism to resolve conflicts, the state at different sites would diverge. For example, the delete operation issued by site S2 would result in the deletion of "a" instead of "b" without transformation. OT allows the execution of operations (O1, O2) immediately after both have been generated locally, and then transforms concurrent operations (if necessary) to ensure state converges at both sites S1 and S2 [9]. Using the same example as shown in Figure 1, the operation O2 is transformed at site S1 so that character "b" is deleted.

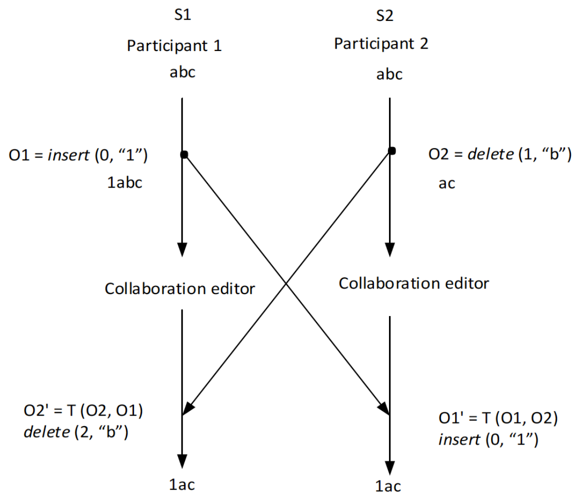

Figure 1. An example scenario for two participants using Operational Transformation.

2.2. The ACE Collaborative Editor

The ACE real-time collaborative editor is the only well-documented, open source project that we could find (http://sourcefrog.net/project/ace/). Note that the term ACE is simply the name for the editor instead of an acronym. The editor is written in Java and it implements the Jupiter algorithm [12]. The Jupiter algorithm is a centralized algorithm for participant coordination and operational transformation. The ACE collaborative editor architecture is shown in Figure 2. The session server component is created when a user first publishes a document. At this point, this user becomes the publisher for the document. The session server manages all interactions between the publisher and participants for the same shared document.

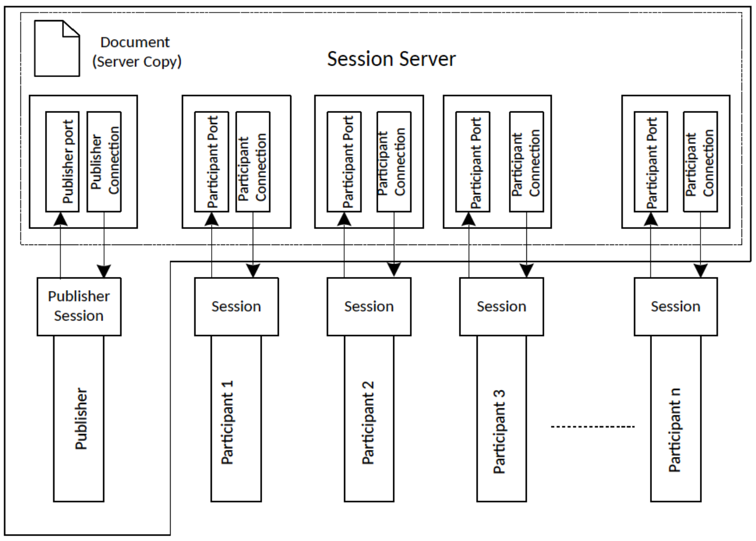

Figure 2. The architecture of the ACE editor with one publisher and multiple participants.

A user can join an editing session by sending the publisher a join request. If the publisher accepts the join request, it notifies all existing participants of the session for the shared document. Similarly, a participant could leave a session by sending a leave request, and all existing participants will be notified for the membership change as well. The publisher can also remove a misbehaving participant from the session and blacklist the faulty participant to prevent it from ever joining again. In addition to managing the membership of the system, the publisher handles update operations submitted by all participants.

The interaction between a participant and the publisher is shown in Figure 3. As shown in Figure 3, the user requests to join a session (step 1) and then the publisher will accept or deny the request (step 2). Upon acceptance, the user requests creating a session (step 3) to participate in the editing of the shared document. The publisher provides the user with a session ID (step 4) and issues a nested invocation to create a document's replica (step 5 and step 6) and sends the replica to the participant (step 7). The participant can then edit the shared document (Step 8). The server copy of the shared document is updated (step 9 and step 10) and the updated document is also exchanged with the participant (step 11). The session can be terminated at any time between the participant and the publisher (step 12 and step 13).

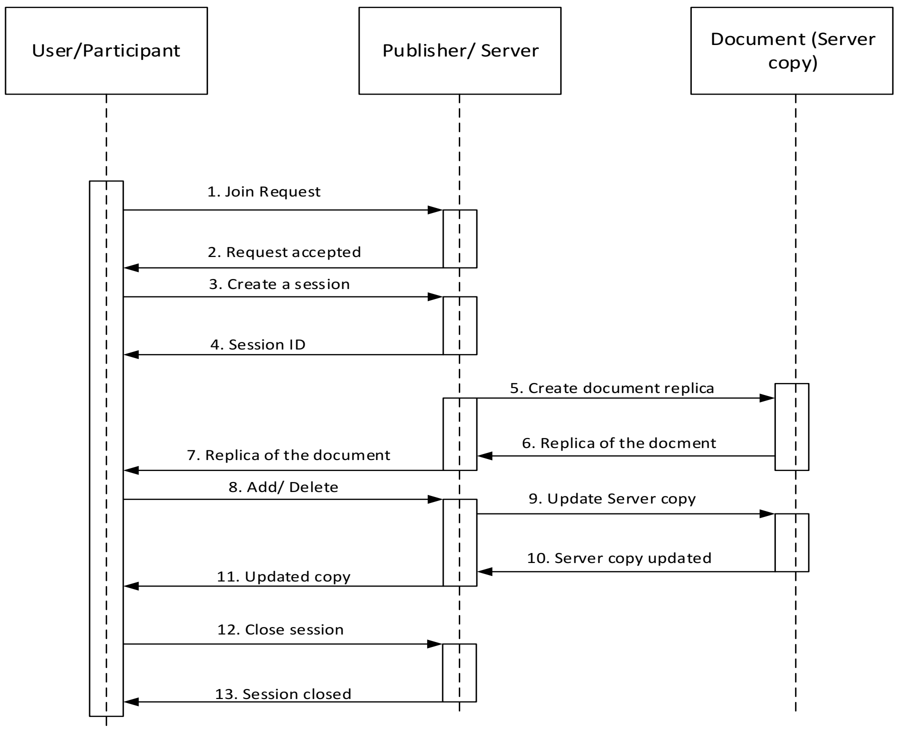

Figure 3. An example editing session with ACE.

2.3. Related Work

Compared with the traditional way of ensuring replica consistency in a replicated system, which is typically based on totally ordering all updates and applying them sequentially, OT algorithms [1,10,11,13] provide a powerful alternative while facilitating concurrent execution of updates. However, they are not designed with fault tolerance in mind.

Besides our own research on this topic [14,15], the limited number of publications on improving fault tolerance for collaborative editing systems all require the use of the crash–fault model [16–18]. Qin and Sun proposed using a primary-backup scheme to tolerate a single crash fault at the server [16,17], where the shared document is maintained only at the server and it is replicated for fault tolerance. Another contribution from [16,17] is the use of periodic checkpointing to reduce the recovery time for a failed client. The state synchronization mechanism in our work also depends on the checkpointing of the shared document. Shim and Prakash [18] proposed to use a stateful group communication system to build fault tolerant groupware applications. The application programming interfaces provided by the group communication system are used to make it easier to manage the shared document and the membership changes at such applications. As a tradeoff, using a proprietary group communication system would make it difficult to deploy the system over the Web. The mechanisms introduced in this paper not only protect the system from both crash and malicious faults, they do so by utilizing the built-in redundancy of collaborative editing systems.

Byzantine fault tolerance has been a hot research area since Castro and Liskov revitalized this field with their seminal work [4]. Numerous BFT algorithms have been proposed [19–29]. Many of these algorithms are designed to protect generic stateful servers against Byzantine faulty server replicas and faulty clients. It is not our goal to develop a competing algorithm to traditional BFT algorithms. Rather, our objective is to develop efficient BFT solutions for collaborative editing systems by using a combination of lightweight mechanisms and by employing traditional BFT algorithms only when they are needed. This is inline with our previous research on other types of systems [7,20,24,25,30].

This article is an extension of our conference paper [14]. Even though we followed the approach as in [14], we have made a number of extensions, including correcting a technical error related to state synchronization, providing a more detailed state synchronization algorithm, defining the correctness properties of the mechanisms with proof, implementing the proposed mechanisms, and reporting the performance evaluation results of the mechanisms. The state synchronization mechanism in our work is in a way similar to the deferred updates idea proposed by Luiz et al. [31]. Both mechanisms require a primary replica to disseminate information to other replicas, and both require a round of a Byzantine agreement (for each transaction in [31], and for each round of state synchronization in our work). However, our mechanism differs from that in [31] in that our mechanism is an optimistic replication mechanism where updates are applied to a local replica immediately and the state synchronization is used to achieve the eventual consistency of different replicas. The deferred updates mechanism in [31] operates within the boundary of a transaction, where updates are not visible to other transactions or clients until the transaction has been committed. Furthermore, our mechanism allows for the flexibility of applying state synchronization for a different number of updates. The number of deferred updates in each transaction in [31] cannot be changed arbitrarily because it is determined by the application semantics. Typically, the number of updates used in our state synchronization mechanism is much larger than that in each transaction.

3. Methods

We first conduct a threat analysis on the real-time collaborative editing system. Based on the threats, we then propose a set of BFT mechanisms to control these threats. The correctness properties and the proof of correctness of our mechanisms are provided in Appendix A.

3.1. Threat Analysis

In this section, we analyze potential threats that could compromise the integrity of a collaborative editing system. We limit our analysis to systems that use centralized coordination such as ACE. As argued in the ACE documentation, centralized coordination is the only way to facilitate operational transformation that is provably correct. Centralized coordination requires the existence of a publisher that carries more responsibility than regular participants (such as user registration and mediation of updates), as we have outlined in the previous section. As a tradeoff, a compromised publisher could impose greater threat to the integrity of the collaborating editing application. Note that the use of a publisher does not necessarily mean that there is a single point of failure in the system. A regular participant could assume the role of the publisher if the current publisher has been found to have been compromised. On the other hand, centralized coordination brings benefits as well. It insulates participants from each other so that a faulty participant cannot directly affect another participant.

In this study, we only consider insider threats, such as threats from compromised participants or the publisher. We do not consider threats launched by external adversaries. Such threats could be launched against any system. Hence, they are out of the scope of this article.

3.1.1. Threats from a Faulty Publisher

The following threats can be imposed by a faulty publisher:

1. Malicious updates: A faulty publisher might introduce malicious updates on the shared document.
2. Partition attack: A faulty publisher may selectively pass on a membership change to a portion of participants with the intention of creating artificial partitions among the participants. The consequence of this attack is that participants in different partitions would have different versions of the shared document.
3. Inconsistent updates: A faulty publisher may selectively relay an update submitted by a participant to a subset of the participants. Again, this attack would cause participants to have different versions of the shared document.
4. Denial-of-service attack: A faulty publisher might launch a denial-of-service attack on any of the participants by refusing to accept a join request.

In all these cases, the integrity of the services is compromised. These types of threats except case 4 can be controlled by our lightweight BFT algorithms. For case 4, the user could notify the system administrator for being refused service by the publisher.

3.1.2. Threats from a Faulty Participant

The following threats may be imposed by a faulty participant:

1. Malicious updates: A faulty participant can inject malicious updates to the shared document. For example, a faculty participant can introduce inappropriate texts or delete texts that should not be deleted from the shared document.
2. Denial-of-service attack: A faulty participant can repeatedly join and leave an editing session with the intention to increase the load on the publisher (the session server to be specific). Handling the join request is an expensive operation for the publisher because the publisher would have to send the current shared document to the new participant on each join. This constitutes a form a denial-of-service attack on the publisher (and hence on the entire application).

Case 1 can be controlled by the proposed BFT mechanisms by detecting malicious updates and by rolling back detected malicious updates. Case 2 can be controlled relatively trivially by implementing some heuristic decision 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.

3.2. The Lightweight BFT Mechanisms

In this section, we present the system model and the main lightweight BFT mechanisms. Our mechanisms accomplish two objectives: (1) ensure a consistent membership; and (2) achieve eventual replica consistency. From the threat analysis, we can see that a faulty publisher could partition the system, hence creating serious problems. To control such threats, we ensure that all membership related operations, such as join and leave, reach a Byzantine agreement before they are executed. This conservative approach is acceptable because such operations must be done sequentially anyway. For normal updates issued by a participant of the system, on the other hand, they must be allowed to proceed concurrently and immediately. Hence, optimistic Byzantine fault tolerance must be used. For this, we follow the "trust, but verify" principle in that we allow the users to proceed as usual most of the time, but periodically synchronize their state.

3.2.1. System Model

We assume that the system is centralized where a centralized algorithm is used to coordinate all 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 a zero-configuration networking mechanism [32] called Bonjour [33]. Bonjour uses multicast Domain Name System (DNS) and DNS service discovery. On the other hand, if the system is deployed on the Internet, 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 users to join the session. We do not assume that the users know each other's contact information unless they have joined the same session.

We further assume that each user has a public/private key pair. The public key is known to all potential users and participants in the same session, while the private key is kept secret to its owner. All messages exchanged in the system are digitally signed to ensure accountability and to prevent spoofing. Furthermore, adversaries have limited computing power so that they cannot break the digital signatures created by nonfaulty users. We assume that the messages can be reliably exchanged between different entities in a session-oriented application, which can be satisfied by using the Transmission Control Protocol (TCP) and by protecting messages sent via TCP with digital signatures.

Finally, we assume that there is sufficient number of users (publisher and participants combined). Given N numbers of users, the system could tolerate up to $f = (N-1)/3$ compromised users. Obviously, N must be greater or equal to 4. This large total number of users is necessary to achieve Byzantine agreement among nonfaulty users.

3.2.2. Consistent Membership

The foundation for correct operation in any distributed system is the agreement on who is part of the system, i.e., every participant of the system should have a consistent view of the current membership. When the publisher receives a join request from a new user, it adds the user to the current membership and initiates a Byzantine agreement on the new membership with all current participants. To reach a Byzantine agreement, the publisher and the existing participants follow the three-phase commit algorithm (referred to as the PBFT algorithm) introduced by Castro and Liskov in [6], as shown in Figure 4. Compared with the PBFT algorithm, there are several differences in our membership algorithm:

- The PBFT algorithm only ensures the agreement on one thing: the binding between a particular sequence number (representing the total order) and an incoming request. In our case, we want to reach an agreement on the new membership. Hence, the sequence number is replaced by a membership operation sequence number, and the message digest of the request is replaced by the full membership set.
- If the publisher could complete the Byzantine agreement step, it approves the membership and notifies the new user about the membership. In the notification message, the full membership information signed by all voting participants is included so that the new participant could participate future group-wise activities, such as new membership changes and state synchronizations. It is important that the membership is signed by every voting participant so that a faulty publisher cannot send a fake membership to the new participant.

We should note that a membership service is essential to traditional group communication systems [34,35]. The membership mechanism described here is different from these traditional approaches because it is built on top of Byzantine agreement algorithms.

If the publisher is faulty and intentionally disrupts the agreement on the membership operation by sending different memberships to different participants, the agreement may not be completed at some or all participants. These kind of attacks can be easily detected during the prepare phase of the Byzantine agreement because all nonfaulty participants broadcast the membership that have received from the publisher to all others. On receiving a conflicting signed membership, a participant suspects the publisher and initiates a view change to elect a new publisher.

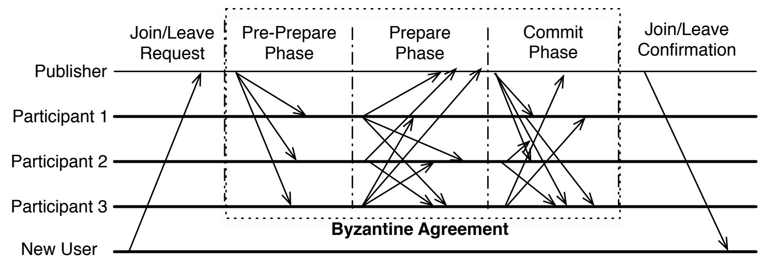

Figure 4. Byzantine agreement on membership changes.

3.2.3. State Synchronization

The state synchronization is the key mechanism to achieve eventual consistency among the participants. The main steps for the mechanism are illustrated in Figure 5. The first step is to determine a synchronization point. Because we assume that centralized coordination is used, the publisher is in the best position to decide when it is time to perform a new round of state synchronization. For example, the publisher could be configured to perform a new round of synchronization for every n updates (e.g., 100 updates) processed. Right after the publisher processed the n-th update since the last synchronization, it issues a state synchronization request to all participants. Because each participant is performing updates on the shared document independently until receiving the synchronization request, it may happen that a participant has already applied one or more local updates by the time the sync request arrives. In this case, the local updates at the participant must be rolled back. This is inevitable.

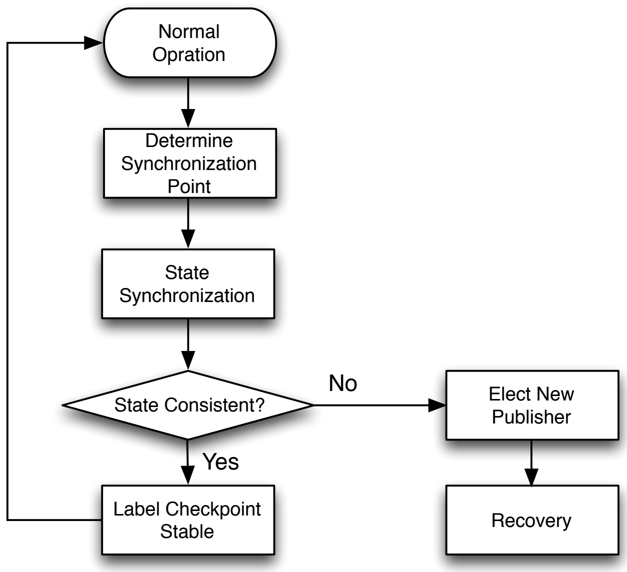

Figure 5. Main steps in state synchronization.

There are two modes of operations for state synchronization: (1) normal operation; (2) recovery. All state synchronization starts with the normal operation. If the publisher is not faulty, state synchronization would run successfully in the normal operation mode. If the normal operation could complete successfully, then a stable checkpoint of the current version of the shared document would be produced for this round of state synchronization and saved on stable storage. This stable checkpoint will be used if recovery is needed in the next round of state synchronization.

Note that, even for the normal operation, a Byzantine agreement is needed to ensure that all nonfaulty participates agree on both the synchronization point and the checkpoint produced in the current round of state synchronization. When the publisher decides that it is time to do a new round of state synchronization, it multicasts a sync-init message to all participants. The message has the form $< \text{SYNC-INIT}, v, r, S, d, i >_{\sigma_i}$, where v is the current view number, r is the round number, which is increased by one for every new round of state synchronization, S is the set of two-dimensional vectors, one per participant, that indicate the last update that should have been applied at each participant, d is the digest of the shared document, i is the site id, and σ_i is the digital signature signed by sender i.

On receiving the sync-init message from the publisher, the participant validates the signature, and checks the validity of the request regarding issues such as whether or not the synchronization point is correct. If the request passes this validation test, the participant checks to see if it has applied any local updates that are logically later than the synchronization point. If so, such local updates are undone first. Then, the participant compares the received digest d with the digest of its local copy of the shared document. If two digests are identical, the participant produces a local checkpoint of the shared document at this synchronization point, saves the checkpoint to the stable storage, and multicasts a sync-prepare message, which takes the form $< \text{SYNC-PREPARE}, v, r, d, R, j >_{\sigma_j}$, where j is the site id of the participant, and R is the sync-init message received by the sending participant. R is necessary for a nonfaulty participant to tell who is at fault in cases when the digest d in the sync-prepare message is different from that of its own.

On receiving the sync-init message together with $2f$ matching sync-prepare messages from different participants (including the one it has sent), the participant multicasts a sync-commit message, which takes the form $< \text{SYNC-COMMIT}, v, r, d, j >_{\sigma_j}$. The publisher also multicasts the sync-commit message once it has collected $2f$ matching sync-prepare messages from different participants. On receiving $2f$ matching sync-commit messages from other participants and the publisher, the participant or the publisher concludes the Byzantine agreement for state synchronization, and label the local checkpoint stable. This concludes the round of state synchronization.

When a participant detects that the digest d included in the sync-init message from the publisher is different from that of its own, it suspects the publisher and initiates a round of view change. Similarly, a participant may find that the digest included in the sync-prepare message sent by some another participant is different from that of its own. This might happen if the any of the following is the case:

- A faulty publisher could send different sync-init messages (for the same round of state synchronization) to different participants with the malicious intent of getting nonfaulty participants to commit to different states;
- A faulty participant sends a sync-prepare message with a different digest from that of the nonfaulty publisher.

It is important for a nonfaulty participant to differentiate the two cases because the publisher should be suspected in the former case, while the publisher should not be suspected in the latter case. The piggybacking of the sync-init message R in the sync-prepare message serves the purpose. If the digest included in R is different from the digest included in the sync-prepare message, then the sending participant is apparently faulty. On the other hand, if the digest in R is identical to the digest in the sync-prepare message, and the digest is different from participant's own digest for the shared document, then the publisher is faulty. The piggybacking of R in the sync-prepare message is an important mechanism to prevent a faulty participant from lying without being detected. Previously, we applied a similar mechanism in the context of distributed transaction commitment [24].

There is yet another scenario that will lead to the suspicion of the publisher. A publisher might attempt to delay the start of a new round of state synchronization, i.e., it fails to launch a new round of state synchronization when it should. A participant could detect this attack easily by summing up the local and remote timestamps in the vector timestamp included in the update message sent by the publisher. If the sum reaches the multiple of the predefined period for state synchronization, the next message coming from the publisher must be the sync-init message. Note that the participant that made the last update before the state synchronization will not have the opportunity to receive this message, but this will not be a problem because it could detect this attack on receiving the next update message sent by the publisher.

On detection of the faulty publisher or a faulty participant, the operation switches to the recovery mode. If the publisher is suspected, participants will elect a new publisher by initiating a view change, as shown in Figure 6. When a participant suspects the publisher, it multicasts a view-change message to all other participants. The message takes the form $< \text{VIEW-CHANGE}, v+1, r, A, P, i >_{\sigma_i}$, where $v+1$ is the next view number, r refers to the current round of state synchronization, i is the sender id, A is the set of updates submitted by participant i together with their respective timestamps, and P is the evidence that the publisher misbehaved, if any.

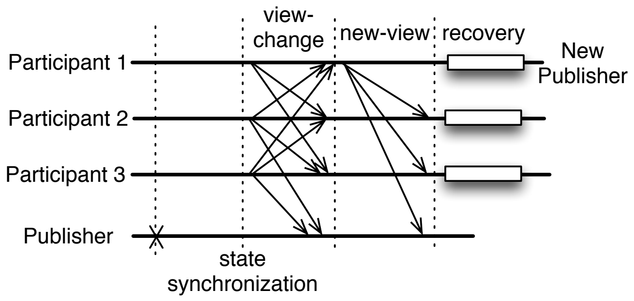

Figure 6. Steps in recovering from a faulty publisher.

On receiving a view-change message, a participant accepts the message and temporarily buffers the message if the message is valid. If the message contains a valid P component, or the participant has received $f+1$ valid view-change messages from other participants, then the participant also multicasts a view-change message. If the publisher in view $v+1$ could receive $2f+1$ valid view-change messages from different participants (including its own), it assumes the leadership and multicasts a new-view message, which takes the form $< \text{NEW-VIEW}, v+1, r, AS, i >_{\sigma_i}$, where AS is the collection of the A sets submitted by the $2f+1$ participants.

When a participant receives a valid new-view message, the leader election is concluded. Before a participant switches to the normal operation mode, however, it has to go through a recovery stage. Unlike other systems where the information included in the $2f+1$ view-change messages is sufficient for recovery, in a collaborative editing system, the updates submitted by all nonfaulty participants are needed for a full recovery. To facilitate recovery, the new publisher will continue collecting the view-change messages until all participants have responded. If a participant fails to send their view-change messages within a predefined period, the new publisher removes that participant from the membership to ensure the liveness of the system. On receiving any of these additional view-change messages that are not part of the $2f+1$ view-change messages, the new publisher extracts the A component and forwards it to all participants.

On receiving the set of updates from every participant in the membership, both the publisher and the participants follow a deterministic algorithm to build the full set of updates. A participant first rolls back its local copy of the shared document to the last stable checkpoint, and then applies this set of updates on the local copy of the shared document.

On detection of a faulty participant, all updates submitted by this participant are to be removed. Since they have already been applied to the shared copy, it is not trivial to cancel their effect. One possible way to achieve this purpose is to first rollback to the last stable checkpoint, and re-apply the updates submitted by nonfaulty participants that are causally preceding to the first operation submitted by the faulty participant since the last state synchronization. Updates that are causally depending on any of the updates submitted by the faulty participant must also be discarded.

3.2.4. Discussion

So far, we have assumed that there is a sufficient number of participants to complete a Byzantine agreement. If this is not the case, then we will have to let the publisher make the sole decision on routine membership changes such as the joining and leaving of a participant. However, periodic state synchronization can be formed even if there is not a sufficient number of participants with some changes to the mechanisms described previously, as explained below.

When it is time to perform a round of state synchronization, the publisher would multicast a sync-init message as usual. A participant would then multicast a sync-prepare message in response if the sync-init is valid. Instead of collecting $2f$ matching sync-prepare messages, a publisher/participant would attempt to collect sync-prepare messages from all participants currently in the membership. In the absence of failure, everyone would be able to collect all the required set of messages, which would conclude the round of state synchronization. Obviously, if a participant fails, the publisher must remove the failed participant(s) before the state synchronization can be completed.

If a state discrepancy is detected, and there is solid evidence that the publisher misbehaved, a view change would take place. The view change algorithm is modified in a similar fashion as that for the state synchronization in that the input from all participants would be required. If a participant is found to have misbehaved, it will be removed from the membership.

Finally, we note that our state synchronization mechanisms incurs minimum blocking in the absence of failures because a local checkpoint is taken as soon as a participant has validated the sync-init message. The participant is free to submit updates immediately after the checkpoint is taken.

4. Results

The proposed mechanisms have been incorporated into the ACE editor. The digital signature and message digest computation are implemented using Java Cryptography Extension. Communication between the publisher and the participants is done via TCP/IP. For Byzantine agreement, we employ the PBFT algorithm [4]. In previous works [20,24,30,36], we have implemented the PBFT algorithms in Java for multiple systems. The implementation was adapted for the ACE editor.

The evaluation is carried out using a testbed consisting of four nodes connected via a Gigabit Ethernet. Each node is equipped with a Core i5-4250U CPU and 4GB of RAM, and runs the Ubuntu 14.04 Linux. One of them initially runs as the publisher and the remaining ones run as the participants. The system is set up to tolerate a single Byzantine faulty node.

Because of the nature of collaborative editing, we are not concerned about scalability and system throughput. Furthermore, we do not evaluate the runtime overhead of the join/leave operations because the latency of such operations would only impact that user that wishes to join and leave without affecting those who are editing the shared document. We report two performance characteristics of our mechanisms: (1) the latency of a round of state synchronization in the absence of faults; and (2) the latency of view change in the presence of a faulty publisher. We experimented with eight different state synchronization period, ranging from 100 to 800 updates per synchronization with 100 increment. Each update is encoded as a tuple of four elements: (1) the character inserted or deleted; (2) the operation type, such as deletion or insertion; (3) the position of the operation in the shared document (which is modeled as an array of characters), and a two-dimensional vector timestamp of the update (one dimension for the local timestamp, and the other for the remote timestamp, indicating how many updates have been submitted by the local and remote sites, respectively). For each configuration, we run 100 times and take the average latency. The ACE editor is modified so that updates are submitted programmatically in a loop instead of manually by a user.

Figure 7 shows the experimental results. In Figure 7a, we show both the blocking time that a user would experience during each round of state synchronization, and the total latency of the entire round of state synchronization. As can be seen, the blocking time is below 10 ms, which shouldn't be noticeable by a user of the system. The total state synchronization latency is about only 20 ms. It may be surprising that there is no apparent dependency between the latency and the synchronization period. However, the lack of dependency is expected because the state synchronization messages only include the digest of the shared document instead of the actual updates submitted by participants. This implies that in the local area network context, we can aggressively perform state synchronization more frequently without incurring excessive runtime overhead while enjoying the benefit of catching a faulty publisher quickly. On the other hand, if the system is deployed on the Internet where users are geographically distributed, the additional delay introduced by the Internet (which is typically around 100 ms) would make it impractical to have frequent state synchronization.

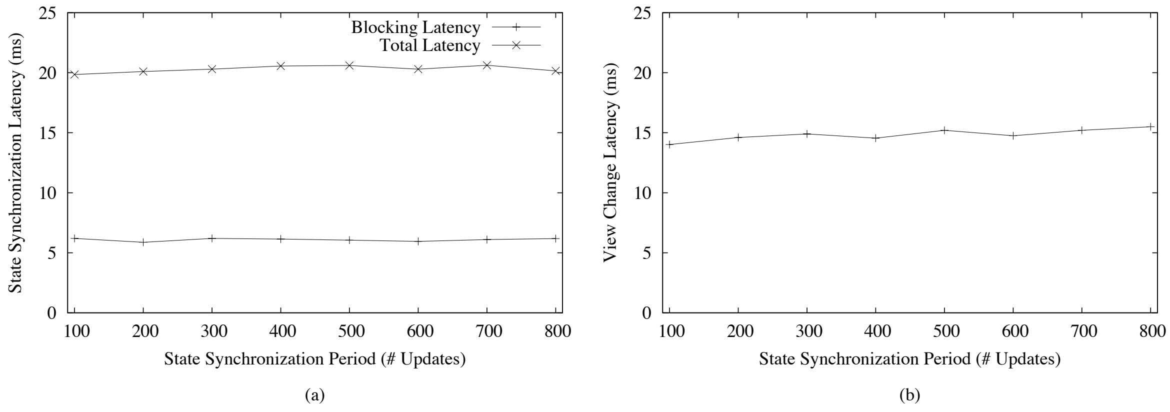

Figure 7. Experimental results. (**a**) state synchronization latency; (**b**) view change latency.

As can be seen in Figure 7b, the view change latency is only about 15 ms and it has very weak dependency on the synchronization period. Even though the message size increases with larger synchronization period, the total length is still small because each update-record is only 16 bytes long. The use of a Gigabit Ethernet ensures that the transmission latency of the messages is below 1 ms even when the synchronization period is 800 updates. This would change if the system is deployed on the Internet, where the upload bandwidth is very limited (in the United States, it is common to have an upload bandwidth as low as 1 Mbps for residential Internet services). For example, if the upload bandwidth is 1 Mbps, the transmission latency for view change messages would range from about 13 ms for 100-update-per-state-synchronization, to about 103 ms for 800-update-per-state-synchronization.

5. Conclusions

In this article, we presented how to enhance the trustworthiness of a real-time collaborative editing application with a set of lightweight Byzantine fault tolerance mechanisms. We followed the principle of optimistic replication [7] by allowing each participant to apply its updates immediately to its local copy and to all remote copies of the shared document. Periodically, a round of state synchronization is conducted to ensure that the states of the participants become consistent. Byzantine agreement is used only to achieve consistent membership and during the state synchronization. We have implemented the proposed mechanisms and integrated them into the ACE real-time collaborative editor. We further characterized the cost of state synchronization and view change of the implemented system with respect to different state synchronization periods. We found that the latency overhead caused by our mechanisms normally does not have negative impact to users.

Acknowledgments: This study is supported by the Graduate Faculty Research Support Program at Cleveland State University. We did not receive funds for covering the costs to publish in open access. We would like to sincerely thank the anonymous reviewers for their insight comments and suggestions on an early version of the manuscript.

Author Contributions: Mamdouh Babi and Wenbing Zhao conceived and designed the experiments; Mamdouh Babi performed the experiments; Mamdouh Babi and Wenbing Zhao analyzed the data; Mamdouh Babi and Wenbing Zhao wrote the paper. Both authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest. The funding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

BFT	Byzantine fault tolerance
PBFT	Practical Byzantine Fault Tolerance
OT	Operational transformation
TCP	Transmission Control Protocol
DNS	Domain Name System

Appendix A

Here, we define the correctness properties of the proposed lightweight BFT mechanisms and provide proof of correctness. The properties are phrased as two theorems.

Theorem A1. *All nonfaulty participants of an editing session see the same membership.*

Proof of Theorem A1. The membership of the system can be changed only with three different operations: (1) a join request; (2) a leave request; and (3) the publisher decides to remove a misbehaving participant. For all such operations, a Byzantine agreement from all current participants is needed. This ensures that, if a nonfaulty participant accepts a membership, all nonfaulty participants accept the same membership. Hence, the theorem holds true. □

Theorem A2. *The state of non-faulty replicas will eventually converge.*

Proof of Theorem A2. Periodically, the publisher and participants synchronize their states. They attempt to reach a Byzantine agreement on the synchronization point as well as the version of the shared document at the synchronization point. If they could reach a Byzantine agreement, the states are proven to be consistent (i.e., not divergent) and no other action is needed. If they fail to reach a Byzantine agreement, then a new publisher is elected. If this new publisher is not faulty and the environment is sufficiently synchronous, a new view change will succeed and the recovery actions will ensure the convergent of the states. This proves Theorem A2. □

References

1. Ellis, C.A.; Gibbs, S.J. Concurrency control in groupware systems. In Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD '89), Portland, OR, USA, 31 May–2 June 1989; ACM: New York, NY, USA, 1989; pp. 399–407.
2. Li, D.; Li, R. An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems. *Comput. Support. Coop. Work* **2010**, *19*, 1–43.
3. Sun, C.; Jia, X.; Zhang, Y.; Yang, Y.; Chen, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput. Hum. Interact.* **1998**, *5*, 63–108.
4. Castro, M.; Liskov, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* **2002**, *20*, 398–461.
5. Zhao, W. *Building Dependable Distributed Systems*; Wiley-Scrivener: Beverly, MA, USA, 2014.
6. Zhao, W. Concurrency Control in Real-Time E-Collaboration Systems. In *Encyclopedia of E-Collaboration*; Kock, N., Ed.; Idea Group Publishing: Hershey, PA, USA, 2008; pp. 95–101.
7. Zhao, W. Optimistic Byzantine fault tolerance. *Int. J. Parallel Emerg. Distrib. Syst.* **2016**, *31*, 254–267.

8. Babi, M.; Zhao, W. Conflicts and Resolutions in Computer Supported Collaborative Work Applications. In *Encyclopedia of Information Science and Technology*, 3rd ed.; Khosrow-Pour, M., Ed.; IGI Global: Hershey, PA, USA, 2015; pp. 567–575.
9. Babi, M.; Zhao, W. Increasing the Trustworthiness of Collaborative Applications. In *Encyclopedia of Information Science and Technology*, 3rd ed.; Khosrow-Pour, M., Ed.; IGI Global: Hershey, PA, USA, 2015; pp. 4317–4324.
10. Sun, C.; Ellis, C. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work, Seattle, WA, USA, 14–18 November 1998; pp. 59–68.
11. Li, R.; Li, D. A new operational transformation framework for real-time group editors. *IEEE Trans. Parallel Distrib. Syst.* **2007**, *18*, 307–319.
12. Nichols, D.A.; Curtis, P.; Dixon, M.; Lamping, J. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95), Pittsburgh, PA, USA, 15–17 November 1995; ACM: New York, NY, USA, 1995; pp. 111–120.
13. Sun, D.; Xia, S.; Sun, C.; Chen, D. Operational transformation for collaborative word processing. In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, Chicago, IL, USA, 6–10 November 2004; pp. 437–446.
14. Zhao, W.; Babi, M. Byzantine fault tolerant collaborative editing. In Proceedings of the IET International Conference on Information and Communications Technologies, Beijing, China, 27–29 April 2013; pp. 233–240.
15. Zhao, W.; Babi, M.; Yang, W.; Luo, X.; Zhu, Y.; Yang, J.; Luo, C.; Yang, M. Byzantine fault tolerance for collaborative editing with commutative operations. In Proceedings of the IEEE International Conference on Electro Information Technology, Grand Forks, ND, USA, 19–21 May 2016; pp. 246–251.
16. Qin, X.; Sun, C. Efficient Recovery Algorithm in Real-Time and Fault-TolerantCollaborative Editing Systems. In Proceedings of the ACM Workshop on Collaborative Editing Systems, Philadelphia, PA, USA, 2–6 December 2000.
17. Qin, X.; Sun, C. Recovery Support for Internet-Based Real-Time Collaborative Editing Systems. In Proceedings of the International Conference on Computer Networks and Mobile Computing (ICCNMC '01), Beijing China, 16–19 October 2001; IEEE Computer Society: Washington, DC, USA, 2001; p. 181.
18. Shim, H.S.; Prakash, A. Tolerating Client and Communication Failures in Distributed Groupware Systems. In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, Madrid, Spain, 2–4 December 1998; IEEE Computer Society: Washington, DC, USA, 1998; p. 221.
19. Amir, Y.; Danilov, C.; Kirsch, J.; Lane, J.; Dolev, D.; Nita-Rotaru, C.; Olsen, J.; Zage, D. Scaling Byzantine fault-tolerant replication to wide area networks. In Proceedings of the International Conference on Dependable Systems and Networks, Philadelphia, PA, USA, 25–28 June 2006; pp. 105–114.
20. Chai, H.; Zhang, H.; Zhao, W.; Melliar-Smith, P.M.; Moser, L.E. Toward trustworthy coordination of Web services business activities. *IEEE Trans. Serv. Comput.* **2013**, *6*, 276–288.
21. Cowling, J.; Myers, D.; Liskov, B.; Rodrigues, R.; Shrira, L. HQ Replication: A Hybrid quorum protocol for Byzantine fault tolerance. In Proceedings of the Seventh Symposium on Operating Systems Design and Implementations, Seattle, WA, USA, 6–8 November 2006.
22. Kotla, R.; Alvisi, L.; Dahlin, M.; Clement, A.; Wong, E. Zyzzyva: Speculative Byzantine fault tolerance. In Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, USA, 14–17 October 2007.
23. Zhang, H.; Zhao, W.; Moser, L. E.; Melliar-Smith, P. M. Design and implementation of a Byzantine fault tolerance framework for non-deterministic applications. *IET Softw.* **2011**, *5*, 342–356.
24. Zhang, H.; Chai, H.; Zhao, W.; Melliar-Smith, P.M.; Moser, L.E. Trustworthy coordination for Web service atomic transactions. *IEEE Trans. Parallel Distrib. Syst.* **2012**, *23*, 1551–1565.
25. Zhao, W. Performance optimization for state machine replication based on application semantics: A review. *J. Syst. Softw.* **2016**, *112*, 96–109.
26. Veronese, G.S.; Correia, M.; Bessani, A.N.; Lung, L.C.; Verissimo, P. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.* **2013**, *62*, 16–30.

27. Behl, J.; Distler, T.; Kapitza, R. Consensus-oriented parallelization: How to earn your first million. In Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, 7–11 December 2015; pp. 173–184.
28. Duan, S.; Peisert, S.; Levitt, K.N. hBFT: Speculative Byzantine fault tolerance with minimum cost. *IEEE Trans. Dependable Secur. Comput.* **2015**, *12*, 58–70.
29. Sousa, J.; Bessani, A. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In Proceedigns of the IEEE 34th Symposium on Reliable Distributed Systems IEEE, Montreal, QC, Canada, 28 September–1 October 2015; pp. 146–155.
30. Chai, H.; Zhao, W. Byzantine fault tolerant event stream processing for autonomic computing. In Proceedings of the IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, Dalian, China, 24–27 August 2014; pp. 109–114.
31. Luiz, A.F.; Lung, L.C.; Correia, M. Byzantine fault-tolerant transaction processing for replicated databases. In Proceedings of the 10th IEEE International Symposium on Network Computing and Applications, Cambridge, MA, USA, 25–27 August 2011; pp. 83–90.
32. Zero Configuration Networking (Zeroconf). Available online: http://www.zeroconf.org/ (accessed on 23 March 2017).
33. Krochmal, M. Rendezvous Is Changing to ... Available online: https://lists.apple.com/archives/rendezvous-dev/2005/Apr/msg00001.html (accessed on 23 March 2017).
34. Reiter, M.K. A secure group membership protocol. *IEEE Trans. Softw. Eng.* **1996**, *22*, 31–42.
35. Correia, M.; Neves, N.F.; Lung, L.C.; Veríssimo, P. Worm-IT—A wormhole-based intrusion-tolerant group communication system. *J. Syst. Softw.* **2007**, *80*, 178–197.
36. Zhao, W. Design and implementation of a Byzantine fault tolerance framework for Web services. *J. Syst. Softw.* **2009**, *82*, 1004–1015.