Integrita: Protecting View-Consistency in Online Social Network with Federated Servers

Sanaz Taheri Boshrooyeh, Alptekin Küpçü, and Öznur Özkasap

Department of Computer Engineering, Koç University, İstanbul, Turkey

Abstract. In the current designs of OSN with a central provider, users' read and write requests over the shared data (e.g., Facebook wall or a group page) are handled via a central OSN provider. However, such centralization comes with view consistency issues where a corrupted provider may serve users with different views of the shared data e.g., by adding, dropping or reordering posts. Integrita provides a data-sharing platform that empowers view consistency relying on N federated servers whose $N-1$ can be malicious and colluding. Users are guaranteed that the servers cannot show divergence view of the shared data (e.g., posts of the group page) to the users (e.g., group members) without being detected. Unlike the state-of-the-art, Integrita enables detection of inconsistency neither by using storage inefficient data replication solution nor by requiring users to exchange their views out of the band. Every user, without relying on the presence of other users, can verify any server-side equivocation regarding her performed operation. We introduce and achieve a new level of view consistency called q-detectable consistency in which any inconsistency between users' view cannot remain undetected for more than q posts. The data-sharing platform of Integrita advances the centralized and distributed counterparts by improving the view-consistency and storage overhead (by the factor of $\frac{1}{N}$ where N is the number of the servers), respectively. Nevertheless, concerning per server storage overhead and cross-server communication, Integrita's overhead is the minimum among all its counterparts.

Keywords: View consistency, q-Detectable Consistency, Strong Consistency, Malicious, Shared Data, Collaborative Data Sharing, Integrity, History Integrity.

1 Introduction

OSNs enable various methods of data sharing like via users' personal walls or social groups. Using the personal wall, a user may share her personal information (e.g., thoughts, images, and videos), with the social connections she authorizes on the OSN, i.e., the friends or followers. In addition to the user being able to continuously update its wall information, her friends or followers may also update her wall by adding post to it e.g., birthday messages, and commenting. A similar data-sharing paradigm appears in the context of social networking

groups like Facebook groups where the members of the group can jointly update the content of the group page by inserting posts.

In the current designs of OSN with a central provider, users' read and write requests over the shared data (being a wall or a group page) is sent to the central OSN server who authorizes the request and acts accordingly. Users' interaction with OSN provider is based on trust that is the server processes the requests honestly and according to the designated instructions. However, in the current practice of OSNs, rather than trust, no technique is deployed to enforce such trustworthy behavior of the OSN server. A corrupted server may add arbitrary content to the shared data and make users accept them as authentic data or hide some posts from some users. As a historical example, in 2012, several bloggers claimed that Sina Weibo, a Chinese OSN, aimed to practice censorship by serving different views of the walls to their followers via hiding some of their posts [9]. Given such historical incidents, it is vital to tackle *view consistency* of the *object* with a practical solution rather than trusting the service provider.

To formalize the problem of *view consistency*, we will use the term shared *object* to indicate a collaborative data-sharing environment (such as a Facebook-like wall or a group-page) on which a set of users are authorized to perform read and write operation. We denote the shared object by $\mathcal{D}$. Each object is comprised of smaller units called *post* which have content and an author. Similar to Frientegrity [8], we assume posts are uploaded to the object one after another hence no concurrency will happen in users write operations. We denote the k^{th} version of $\mathcal{D}$ by $\mathcal{D}_k = \{post_1, \cdots, post_k\}$ namely, an ordered sequence of k posts. Likewise, the view of a user u toward the i^{th} version of $\mathcal{D}$ is comprised of a sequence of i posts seen by that user i.e., $View_i^u = \{post'_1. \cdots post'_i\}$. The view consistency concerns two aspects of users' views. First, to ensure that the corrupted storage provider cannot forge any post i.e., all the $post'_j \in View_i^u$ are issued by authorized users. This can be immediately addressed by deploying digital signatures. The second aspect regards the *history integrity* of an *object* $\mathcal{D}$ which is less recognized and studied in the literature. This second property assures that the view of all the users (obtained through their interaction with the corrupted provider) contains an identical and intact sequence of posts i.e., no post is dropped or misplaced. More formally, for the i^{th} version of object, for every authorized user u and for all $j \in [1, i]$, $post'_j$ is equal to $post_j$ where $post'_j \in View_i^u$, and $post_j \in \mathcal{D}_i$.

Related Work: The view consistency problem is addressed in the literature by two types of solutions: communication-based solutions which are sought by the centralized architectures, and replication-based solutions deployed by the distributed designs. We elaborate on each solution type next.

In a centralized architecture with non-communicating users, the best achievable level of view consistency is fork consistency [30], first defined by [20]. The fork-consistency is a weaker form of view-consistency in which a corrupted provider is able to split users into disjoint sets (to fork them) and serve each set with a distinct view though the provider is forced to serve each set with a consistent view of the operations performed by the users of the same set. Identification

of the forked views can only happen through users' communication. That is users must regularly communicate their views of the *object* (e.g., a wall) with all the other authorized users (e.g., friends) to catch any view inconsistency. This approach would not be practical considering that a user of an OSN like Facebook has 338 friends on the average[1]. Hence, each user needs to communicate with almost 338*338= 114244 other users to monitor the view consistency of her wall and her friends' walls. Addressing view consistency using communication-based solution is sought in the context of secure OSNs [8, 10], and cloud computing [5, 4, 19, 2].

The replication-based solutions are deployed in peer-to-peer OSNs [22] (as a distributed OSN), Authenticated data structures (ADS) [11, 12, 24, 28, 13, 23] as well as Byzantine fault-tolerant protocols [17, 6]. The idea is to designate multiple entities for the storage of the *object* and let all the read and write operations happen through all of them. In specific, the shared object (or some authenticated-metadata associated with it) must be replicated on $f + 1$ entities considering f of them may act maliciously. Having only one honest repository suffices to always retrieve the intact content of the object. Replication based solutions are not efficient concerning the storage overhead since f extra copies of the object must be stored in the OSN.

Integrita: In Integrita, we aim to achieve the best of both aforementioned solutions: a method to achieve view consistency which is replication-free as well as communication-free, and an approach where users do not have to communicate their views out-of-band. In particular, N federated servers run by multiple authorities are utilized, and the storage of shared object is split among them (rather than replicated). Each server gets to serve only a part of the shared object which has no overlap with the parts stored by other servers. This way, we cope with the storage overhead imposed by replication-based proposals as only one instance of the shared object (and its associated meta-data) is maintained in the entire design. We let servers be malicious/Byzantine entities who may act arbitrarily, collude, compromise the view consistency by dropping, tampering with, and forging posts. Nonetheless, our approach guarantees that as long as one server does not collude with the other servers, the view consistency is preserved. We assume that users shall act honestly and tend to achieve a consistent view. A similar assumption is sought in prior studies [7] as well. Note that in Integrita, we are not concerned about the privacy of the posts; one can address it using the well-practiced techniques like encryption [1, 16].

Integrita provides the following features.

- **q-Detectable-Consistency:** In Integrita, we introduce a new level of view consistency called *q-Detectable-Consistency* in which the views of users toward the *object* (i.e., wall) cannot diverge for more than q sequence of posts without detection. That is, if a user uploads a post, either her post correctly becomes a part of the shared object and being seen consistently by all the other users, or she can catch any inconsistency within the next q posts. The

[1] https://www.brandwatch.com/blog/facebook-statistics/

value of q depends on the total number of servers and the number of posts on the shared object at the time of write operation. A thorough analysis of this relation is provided in Section 3.3. Moreover, we provide a formal definition of q-detectable consistency together with a security proof in Section 4.1.

- **Communication-free:** In contrast to the fork-based systems, our fork detection mechanism relies neither on the users' collaboration in sharing their views in each operation nor an out-of-band communication. Every user is able to verify any server-side equivocation regarding her performed operation, alone (without relying on the presence of other users).
- **Replication-free:** Our solution for view consistency is storage efficient as we do not replicate the shared object over all the servers. That is, one copy of the *object* is present in the entire system and each server retains only (an identical) portion of it. Our numerical analysis asserts that by using Integrita, an OSN like Facebook with 2.3 billion monthly active users [2] saves up to 2344 Terabyte storage per year (deploying 20 servers) compared to the replication-based approach.

  Note that each of the N storage providers is modeled as a data-center that would take care of a portion of the data assigned to it. While Integrita does not depend on data replication to achieve view consistency, this does not contradict with the replication of data for the sake of *availability*. Namely, each data-center shall deploy its replication mechanism to maintain the availability of the data assigned to it. However, due to Integrita, the amount of data assigned to each data-center is $\frac{1}{N}$ of the data that would be otherwise assigned by using a replication-based solution.
- **Efficient verification:** In Integrita, each read and write operation is associated with a proof of correctness which must be verified by the user. While the creation and transmission of proof in Integrita are handled in a distributed fashion, resultant overhead for users and the servers concerning the amount of data transmission, the communication and computation complexity is identical to the centralized fork-consistent counterparts [9, 10] (while Integrita enforces a higher level of consistency). In Section 4, we further discuss that distributing the storage among multiple servers not only does not degrade the experience of user's interaction with the system concerning the performance but also lowers the computational and storage overhead on each server (compared to both centralized design and the replication-based proposals).
- **Cross-server Communication-free**: While the storage of the shared object is distributed among N servers, servers do not need to communicate or to coordinate to resolve the users' read and write operations. Instead, all the communication happens solely between the users and the servers.

[2] https://www.statista.com/statistics/264810/number-of-monthly-active-Facebook-users-worldwide/

2 System Model

2.1 Model

Integrita is comprised of N *OSN servers* denoted by S_1,...,S_N (each operated by a distinct authority), a set of users $U_1, ..., U_T$ with read/write access to a shared *object* that is stored at the servers side. We assume all the users have an identical read/write access, though one can enforce more fine-grained access control using the technique proposed by [9]. The N servers are responsible to store the *object*s, serve the users' read and write requests.

The shared *object* is comprised of an ordered sequence of posts as well as is associated with an authenticated data structure that is to be kept at the servers' side. The storage of ADS and the object are divided among the servers where each server only holds a portion. This way, we avoid the space inefficiency of the replication by trading the strong consistency with the q-detectable consistency (in which the inconsistency may happen but would not last for more than q posts i.e., the inconsistency is detectable). Hence only one copy of the object and the associated ADS exist in the entire system.

To monitor the trustworthiness of servers in serving all the object posts consistently to every authorized user, namely, preserving view consistency, each user maintains a local data structure that mirrors the state of that object at the time of reading/writing. Each time that the user reads the object, she has to check whether her local state is consistent with the current state of the object. To ensure q-detectable consistency, users also need to audit their updates on the profile at a certain point after their write operation to ensure that the update has become accessible to all the other users.

2.2 Security Goal

The security goal of Integrita is to achieve q-detectable consistency in which users are able to verify any server-side equivocation regarding their performed operation alone and without relying on the presence of other users. In other words, in a q-detectable consistent system, any inconsistency between users' views cannot remain undetected for more than q posts.

2.3 Adversarial Model

The servers are untrusted hence may act maliciously to compromise the integrity of profile history. This includes dropping a post, reordering posts, and showing users a different subset of posts. We assume that out of N servers, at least one server does not conspire with the rest of the servers. We treat confidentiality as an orthogonal issue to be addressed by encrypting the content of posts. Thus, our sole objective is protecting the users' view consistency.

3 Integrita System Design

In Integrita, we make use of an authenticated data structure called *history tree* to represent the shared *object* and to enable verifiable write and read operation for the users. That is, each read and write operation is associated with a proof through which the user can verify the authenticity of the operation result. The details of *object* representation is provided in Section 3.1. Representing a shared *object* using a *history tree* does not suffice to provide view consistency. We further discuss in Section 3.2 how to distribute the storage of shared *object* among N servers.

3.1 Shared *object* representation

The shared *object* $\mathcal{D}$ is treated as an ordered sequence of posts $\mathcal{D} = \{post_1, ..., post_M\}$. Each post shall be signed by its issuing user and can contain any type of data e.g., text or image. Concerning privacy, one can assume the content is encrypted and the decryption key is provided to all the other users.

The shared *object* is additionally attached to an authenticated data structure called *history tree* which is initially introduced by [7]. A history tree is an append-only data structure modeled by a variant of the Merkle hash tree. In Integrita, the leaves of the tree hold the hash of each post $post_i$. The intermediate nodes and root node store the hash of their children. In such a structure, the root essentially covers the entire content of the tree. The new posts can freely be added as the leaf nodes to the right side of the tree. For each newly added post, the value of intermediate nodes and the root shall be recalculated. A sample of history tree for a shared *object* $\mathcal{D}$ with 4 posts is provided in Figure 1. Figure 2 represents the same tree after the insertion of $post_5$. We use the term of *Tree digest* or TD for short to refer to the history tree root and we write TD_i to indicate the content of the root after insertion of i^{th} post. We refer to the shared *object* with i posts as the i^{th} version of the shared *object*.

The history tree exhibits the following properties that are fundamental inefficiently preserving view consistency.

- Every tree digest TD_j uniquely defines a distinct ordered sequence of j posts. That is, for two identical sets of posts each with a different order e.g., $\mathcal{D}_3 = \{post_1, post_2, post_3\}$ and $\mathcal{D}'_3 = \{post_2, post_1, post_3\}$, their associated tree digests TD_3 and TD'_3 end up completely different. This is due to collision resistant property of the underlying hash function.
- Proof of membership: The occurrence of a particular post $post_i$ at position i in a tree digest TD_j where $i <= j$ is efficiently verifiable in $O(log(j))$. The proof of membership includes the sequence of values stored at the siblings of the nodes (indicating whether it is a left or right sibling) on the path from the leaf node storing $post_i$ to the root TD_j. Given the proof, one can recompute the root as TD'_j and compare against TD_j. For example, as shown in Figure 3, to prove that TD_4 contains $post_3$ as its 3^{rd} post, the

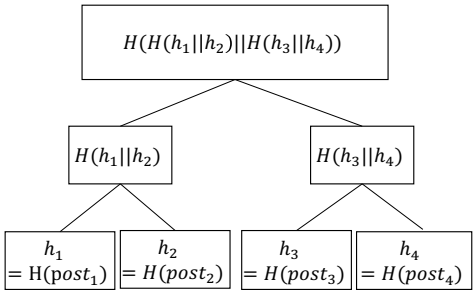

Fig. 1. A history tree constructed for the *object* with 4 posts.

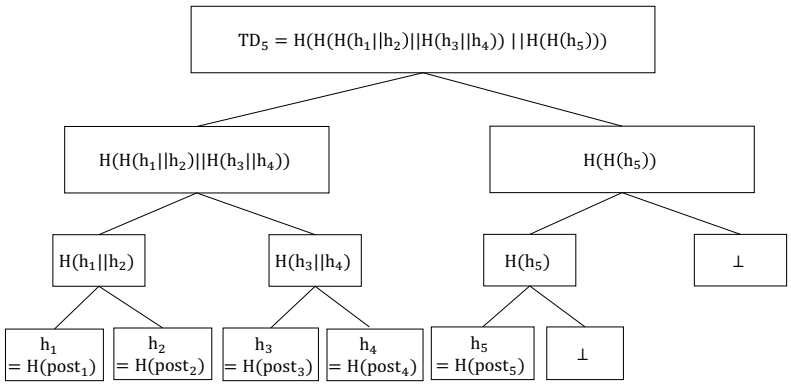

Fig. 2. The tree of Figure 1 after the insertion of $post_5$.

proof includes h_3, h_4, $H(h_1||h_2)$ and $H(H(h_5))$. The tree digest TD'_5 can be reconstructed recursively from the values included in the proof. If the computed value TD'_5 and the given tree digest TD_5 match, then $post_3$ is the 3^{rd} post of the *object*.

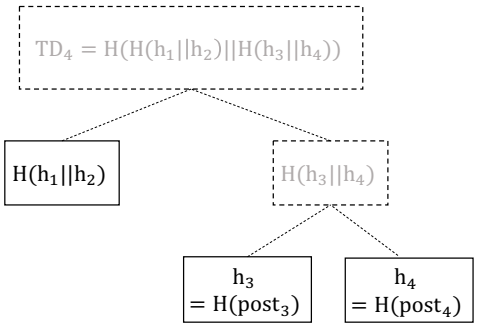

Fig. 3. The membership proof of $post_3$ for version 4^{th} of the shared *object*.

– Incremental Proof: Given two different tree digests TD_i and TD_j of the same shared *object*, where $i < j$, one can check whether the two tree digests make consistent claim about the past posts namely, whether TD_i and TD_j share the same history regarding $post_1, \cdots, post_i$. The incremental proof between version 2^{nd} and version 5^{th} of a shared *object* is shown in Figure 4. Let TD'_2 indicate the tree digests computed using the given proof. If $TD'_2 = TD_2$ then the incremental proof asserts the consistency of TD_2 and TD_5.

3.2 Distributed storage of the shared *object*

For the distributed storage of the shared *object* and its associated history tree, we proceed as follows. First, we define an insertion path of post i to be nodes of history tree whose hash values get altered while inserting post i to the tree. Figure 5 illustrates the insertion path of posts $1-4$ each with a different color. $I_{i,j}$ refers to the j^{th} node at level i. Since Merkle trees support logarithmic path lengths from the root to the leaves, the insertion path of i^{th} post will consist of $\lceil log(i) \rceil + 1$ nodes. For example, the insertion path $post_1$ is comprised of only one node $I_{1,1}$ whereas the insertion path $post_2$ consists of two nodes $I_{1,2}$ and $I_{2,1}$.

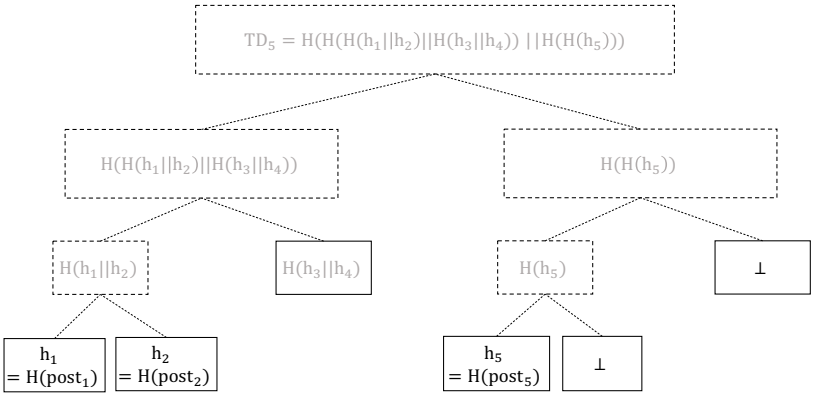

Fig. 4. The incremental proof of the 2nd version of the shared *object* to the 5th version. The solid rectangles represent the proof. The gray parts are computable given the proof parts.

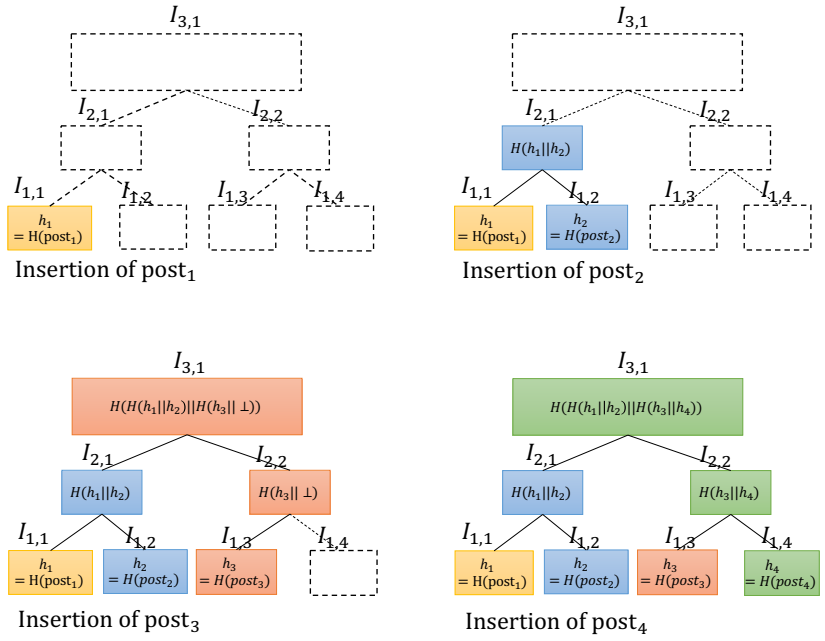

Fig. 5. Insertion path of $post_1$, $post_2$, $post_3$, and $post_4$. Each insertion path indicated by a distinct color.

Note that insertion paths of $post_3$ and $post_4$ have two nodes in common namely, $I_{2,2}$ and $I_{3,1}$. However, the value of each of these nodes at the time of insertion of $post_3$ and $post_4$ are different e.g., $I_{2,2}$, on the insertion path of $post_3$ contains $H(h_3||\perp)$ whereas its value changes to $H(h_3||h_4)$ after insertion of $post_4$. Following this intuition, in Integrita, we treat each of these nodes separately and address them based on their location on the insertion path of each post. That is, each node is addressed with a pair of integers (i, l) as $N_{i,l}$ where i indicates post number and l stands for the level of node on the insertion path. The history tree of Figure 5 under the new addressing is demonstrated in Figure 6. Under the new addressing, node $I_{2,2}$ corresponds to $N_{3,2}$ and $N_{4,2}$ indicating its distinct values at the insertion of $post_3$ and $post_4$.

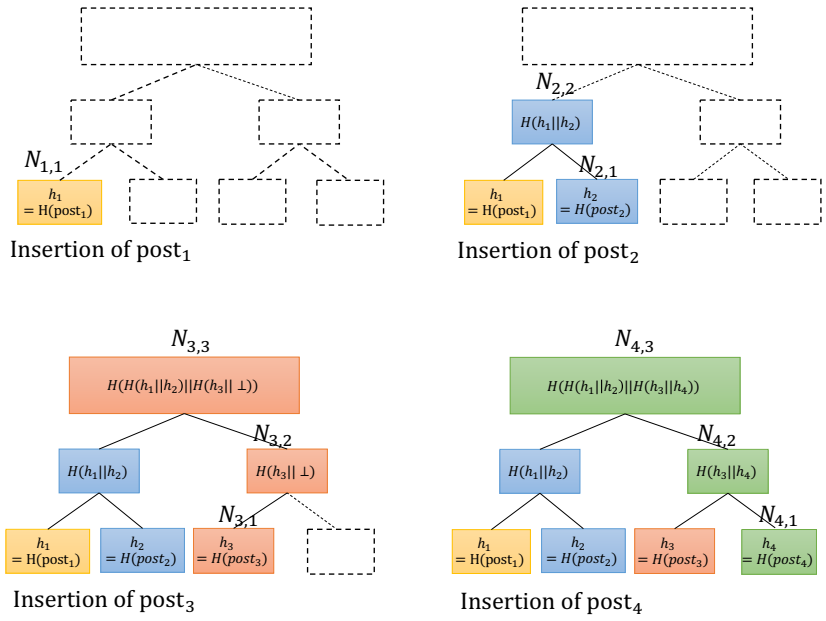

Fig. 6. Insertion path of $post_1$, $post_2$, $post_3$, and $post_4$. Each insertion path indicated by a distinct color. Each node is addressed with a pair of integers (i, l) as $N_{i,l}$ where i indicates post number and l stands for the level of node on the insertion path.

We further define a labeling function L to convert these pairs to a distinct numerical value. $L : \{1, ..., M\} \times \{1, ..., log(M)\} \rightarrow \{0, 1\}^*$ is a deterministic labeling function which receives the pair of (i, l) as defined above and returns back an integer label as given in Equation 1. We labeled nodes of Figure 6 and

showed the result in Figure 7. We write $N_{i,l}$ and $N_{L(i,l)}$, interchangeably, e.g., $N_{4,2}$ and N_8 refer to the same node i.e., $N_{4,2} = N_8 = H(h_3||h_4)$.

$$L(i,l) = l + \sum_{j=1}^{i-1}(\lceil log(j) \rceil + 1) \tag{1}$$

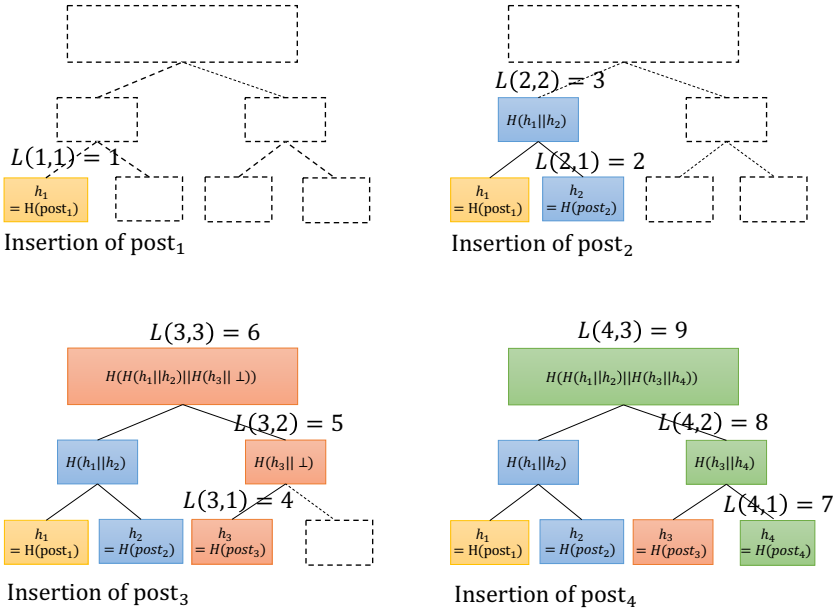

Fig. 7. The labeling of insertion path of $post_1$, $post_2$, $post_3$, and $post_4$ using the labeling function L. Each insertion path indicated by a distinct color. The label of each node is indicated above it.

Given the above labeling mechanism, the storage of each labeled node shall be assigned to a distinct server circularly. For this, we define a function F that receives the label of the node as *label* and returns the index of the server which is responsible for that node. N is the total number of servers.

$$F(i,l) = [L(i,l) \ mod \ N] + 1 \tag{2}$$

For example, in a system with 3 servers, the storage of $N_3, N_6, N_9, \cdots$ are given to the first server S_1. Nodes $N_1, N_4, N_7, \cdots$ are given to S_2 whereas S_3 gets to serve $N_2, N_5, N_8, \cdots$. Say it differently, the assignment of nodes to the servers

follow a circular pattern where $\{N_1 \rightarrow S_1, N_2 \rightarrow S_2, N_3 \rightarrow S_3, N_4 \rightarrow S_1, N_5 \rightarrow S_2, N_6 \rightarrow S_3, \cdots\}$, $\rightarrow$ indicates the assignment. In section 5, we discuss how we enforce q-detectable-consistency using the circular distribution of storage of history tree among the servers.

Following our labeling method, we define and distinguish three types of nodes of the history tree.

- **Tree digest**: The root of tree after insertion of each post is called tree digest. In figure 6, nodes $N_{1,1}, N_{2,2}, N_{3,3}, N_{4,3}$ all represent the tree digests which are roots of the tree at the insertion time of $post_1, post_2, post_3$, and $post_4$, respectively. Given a post number i and the level of node on the insertion path as l, one can identify whether the node is tree digest if Equation 3 holds:

$$l = \lceil log(i) \rceil + 1 \tag{3}$$

- **Full node:** A full node is a node whose left and right sub-trees are full i.e., insertion of further posts will not alter the value of a full node. A node on the l^{th} level of insertion path of i^{th} post is full if the following relation (Equation 4) is met:

$$i \ mod \ 2^{l-1} = 0 \tag{4}$$

  In Figure 6, nodes $N_{1,1}, N_{2,1}, N_{3,1}, N_{4,1}, N_{4,2}$ and $N_{4,3}$ are all full.
- **Temporary node:** Nodes whose left or right sub-trees are not full are called temporary node. We call them temporary since the insertion of further posts will change their hash values. For example, nodes $N_{3,3}$ and $N_{3,2}$ in Figure 6 are temporary as there is an empty node in their right sub-trees corresponding to $post_4$. In general, $N_{i,l}$ is a temporary node if Equation 5 holds:

$$i \ mod \ 2^{l-1} \neq 0 \tag{5}$$

  The content of a temporary node can be reconstructed given the value of the highest full node in its left and its right sub-trees. For example, as shown in Figure 7, the node N_5 as a temporary node can be reconstructed using the highest full node of its left sub-tree which is N_4 and the highest full node in its right sub-tree which is $\perp$. Likewise, the content of N_6 can be correctly recreated using N_3 and its highest full node at its right sub-tree which is N_4 i.e., $N_6 = H(N_3||N_5) = H(N_3||H(N_4|| \perp))$. Due to this property, the storage servers of temporary nodes will not save their hash values. Instead, the responsible servers just maintain some state information about the inserted post (see section 3.3).

Fetching proofs in a distributed manner: In Integrita, there is no central entity holding a global view of the shared *object* and its history tree. As such, dislike the centralized system where the server would create the membership and incremental proofs on its own, in Integrita, the user herself is responsible to determine the nodes on the proof path (using the labeling algorithm given in

Equation 1) and their associated storage providers (using Equation 2) and fetch the hash values. Once hashes are fetched, the user can check the correctness of the proof as normal.

For the ease of explanation, we define the following function

$$\{(S_{F(p,l)}, (p,l))\} = ProofPath(v, R = \{i, \cdots, j\}) \tag{6}$$

that receives a version number v of the shared *object*, and a set $R = \{i, \cdots, j\}$ of a range of post indices. The function returns a set of pairs $(S_{F(p,l)}, (p,l))$ where $N_{p,l}$ is a node holding a value of the membership proof of $\{post_i, \cdots, post_j\}$ and $S_{F(p,l)}$ is the index of the corresponding storage provider. For example, one may call $ProofPath(7, \{5,6\})$ to find out the nodes located on the membership proof of $post_5$ and $post_6$ with respect to the version 7^{th} of the shared *object*. The proof, as illustrated in Figure 9, includes the values of N_9, N_{10}, N_{14}, and N_{18} and their corresponding storage servers S_1, S_2, S_3, and S_4. Note that since the temporary nodes are not saved in the system, the proof includes N_{18} instead of N_{19} which is a temporary node; given N_{18} the calculation of N_{19} is immediate. As we stated before, for a temporary node, its highest full node in its right and left sub-trees shall be fetched instead.

Note that an incremental proof between i^{th} and j^{th} version of the history tree involves the membership proof of $post_i$ and $post_j$. Thus, the function $ProofPath$ can be also used to find the nodes and the servers holding values of incremental proof. For example, finding the nodes on the incremental proof between 2^{nd} and 5^{th} version of the *object* requires a call to $ProofPath(5, \{2,5\})$.

We additionally consider the existence of the following two functions that shall be run by the users:

- $True, False \leftarrow INCR.VF(TD_i, TD_j, proof)$: Given two tree digests TD_j and TD_k where $i < j$ it verifies whether $proof$ is a correct incremental proof between TD_i and TD_j. We write $TD_i \rightarrow TD_j$ to indicate that there exists an incremental $proof$ for which $INCR.VF(TD_i, TD_j, proof))$ returns $True$.
- $True, Fale \leftarrow MEMBERSHIP.VF(TD_j, i, post_i^{'}, proof))$: Given $proof$, the function checks whether the *object* with tree digest TD_j has $post_i^{'}$ as the i^{th} post. This function can further accept multiple posts $MEMBERSHIP.VF(TD_j, \{i, \cdots, k\}, \{post_i^{'}, \cdots, post_k^{'}\}, proof))$ and checks their memberships with respect to TD_j. We write $post_i \in TD_j$ to indicate that there exists a membership $proof$ for which $MEMBERSHIP.VF(TD_j, i, post_i, proof))$ returns $True$.

3.3 Construction

Authorized users (with read and write access to the shared *object*) are associated with a signature key pair. $FList = \{(U_j, vk_{U_j})\}_{j \in [1,T]}$ shall contain the username U_j and the verification key vk_{U_j} of each user. T is the total number of authorized users. $FList$ is publicly available to the servers and the authorized users. Also, posts on the *object* are all encrypted using an encryption key ek. The

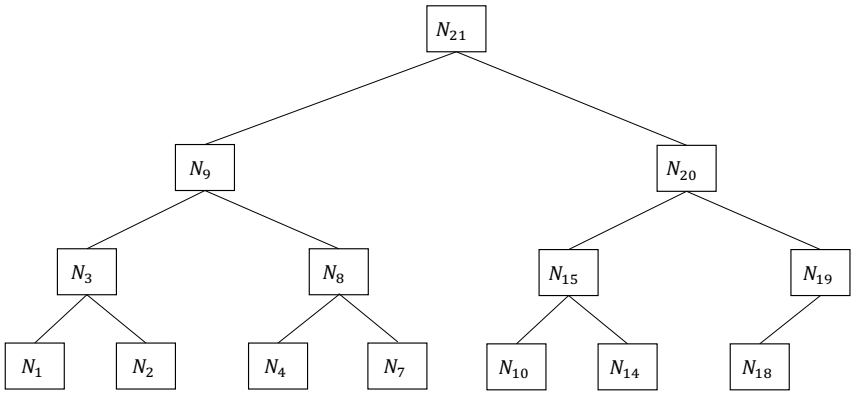

Fig. 8. Version 7^{th} of the shared *object*. The full nodes as well as the nodes on the insertion path of the last post i.e., $N_{18}, N_{19}, N_{20}, N_{21}$ are shown.

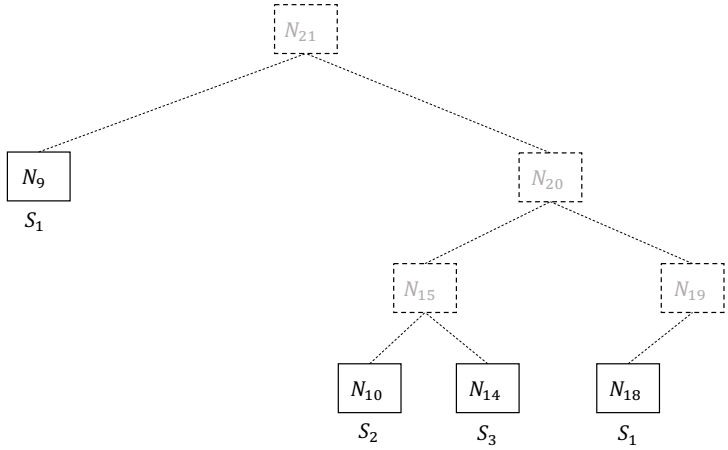

Fig. 9. Nodes located on the membership proof of 5^{th} post concerning the 7^{th} version of the shared *object*. The solid rectangles represent the nodes included in the proof.

corresponding decryption key is given to the authorized users. Note that for the sake of simplicity, we assume that the exchange of $FList$ and the encryption key pair among the users happens out of band, however, one may use the method proposed by [9] to further outsource the storage of $FList$ and management of users access to the servers side. Also, for the ease of explanation, we assume that the set of authorized users is static which can be extended to a dynamic version by deploying the proposal of [9].

Each server has a signature key pair and a unique index in the range of $[1, N]$. Servers publicize the ordered list $SList = \{S_i\}_{i=1:N}$ where each server S_i's index and verification key is accessible through $S_i.index$ and $S_i.vk$, respectively. For simplicity, we assume that the index of each server corresponds to its position in the list i.e., $S_i.index = i$. Besides, the definition of the hash function H to be used in the history tree is publicly available. Each server also sets up a database DB to store the parts of shared *object* and the history tree which is responsible for. Also, each server keeps track of the label of the last seen node as a tuple of (p, l), where p indicates the post number and l is the level of the node, in a local variable *Status* which gets updated once a write operation takes place at that server.

Servers are accessible to the users through three different function calls $Write$, *Read*, and *GetStatus*. Users communicate with servers utilizing these function calls to handle their read and write requests. Namely, users interact with the servers through four protocols *Create object*, *Update Status*, *Read*, and *Write*. We consider an authenticated channel between users and servers. We elaborate on the function calls and protocols below.

Throughout our description, we distinguish between the data generated or operation performed by a server and a user using S and U subscript. That is, we write σ_{U_i} to indicate a signature generated by the i^{th} user. Likewise, $Sign_{U_i}(.)$, and $Verify_{U_i}(.)$ mean the execution of $Sign$ and $Verify$ algorithms using the signature key and verification key of i^{th} user, respectively. Following the same pattern, we will have σ_{S_i}, $Sign_{S_i}(.)$, and $Verify_{S_i}(.)$ for the i^{th} server.

Server-side function calls: Each server is available to the users through three function calls $Write$, *Read*, and *GetStatus* that are explained below.

1. S_j**.Write**$(U_i, (p, l), in = (N_{p,l}, post, \sigma_{U_i}))$: User U_i calls this method to upload the tuple $in = (N_{p,l}, post, \sigma_{U_i})$ to be recorded for the node at the l^{th} level of the insertion path of p^{th} post. $N_{p,l}$ refers to the hash value of the node in the history tree. *post* carries the content of a post and σ_{U_i} is a user-generated signature. Depending on the type of node i.e., tree digest, full or temporary, one or all of these fields might be empty. If the inserted node is a leaf node, then it is associated with the content of a post i.e., *post* and a user-side signature σ_{U_i} over $N_{p,l}||p$. Likewise, the tree digests should be associated with the user-side signature. However, for the temporary nodes, all the fields of *in* are empty.
   The details of $Write$ procedure is shown in Algorithm 1. Firstly, the server needs to check whether the write operation is coming from an authorized user ($U_i \in FList$), the server is the corresponding storage provider of the

Algorithm 1 S_j.Write(U_i, (p,l) , $in = (N_{p,l}, post, \sigma_{U_i})$)

1: **if** $U_i \in FList$ AND $F(p,l) = S_j.index$ AND $F(p,l) - F(S_j.Status.p, S_j.Status.l)=N$ **then**
2: **if** $N_{p,l}$ is a leaf node AND ($H(post) \neq N_{p,l}$ OR $Verify_{U_i}(N_{p,l}||p, \sigma_{U_i}) \neq$ accept) **then**
3: Return Not Verified
4: **end if**
5: **if** $N_{p,l}$ is a tree digest AND $Verify_{U_i}(N_{p,l}, \sigma_{U_i}) \neq$ accept **then**
6: Return Not Verified
7: **end if**
8: Insert $(U_i, (p,l), in)$ into DB
9: **if** $S_j.Status.p \neq p$ **then**
10: Remove the user signature σ_{U_i} $\forall$ $TD_i \in S_j.DB \setminus TD_1$
11: **end if**
12: $S_j.Status$=(p,l);
13: **if** $N_{p,l}$ is a tree digest **then**
14: Return $Sign_{s_j}(N_{p,l})$
15: **else**
16: Return *accept*
17: **end if**
18: Return **Res**
19: **end if**

intended node ($F(p,l) = S_j.index$), and if the node is the next node that the server is expected to receive (i.e., $F(p,l) - F(S_j.Status.p, S_j.Status.l)$=$N$); Note that servers hold $Status$ variable to keep track of their last seen post in the system. The last equality check is correct since the storage assignment of nodes to the servers is circular hence the labels of two consecutive nodes received by a server are N distant. If all the checks passed correctly, the server proceeds as below. If the node is a leaf node or a tree digest (line 2 and line 4), then the server must authenticate the user-side signature (lines 2-7). Upon successful verification, the server inserts the tuple in into the DB (line 8). Also, the server can remove all the user side signatures for the tree digests TD_i which refer to prior posts except the first post (line 9-11). This removal has a significant impact on the storage complexity where at any point in time there will be N signed tree digests saved in the system (rather than all of the signatures of the tree digests). Note that if the node is a temporary node then no data will be recorded for it (as in is empty)). The server updates its Status variable (line 12). If the inserted node is a tree digest, the server must sign the node (lines 13-15) and responds to the user accordingly. Otherwise, the server only acknowledges the success of the write operation (lines 16-18).

2. S_j.***Read***(p, l): Algorithm 2 demonstrates the read procedure. This function receives the index of the node in the history tree i.e., p as the post number and l as the level of the node in the insertion path of p^{th} post. The server checks whether it is responsible for the storage of the requested node i.e.,

Algorithm 2 S_j.Read(U_i, p, l)

Output: $(record = (U_i, (p, l), (N_{p,l}, post, \sigma_{U_k}), \sigma_{S_j})$
1: **if** $U_i \notin FList$ OR $F(p, l) \neq S_j.index$ **then**
2: Return $\perp$
3: **end if**
4: $record = DB.get(p, l)$
5: Parse $record$ as $(*, (p, l), N_{p,l}, *, *)$
6: **if** $N_{p,l}$ is tree digest **then**
7: $\sigma_{S_j} = Sign_{S_j}(N_{p,l})$
8: **else**
9: $\sigma_{S_j} = \perp$
10: **end if**
11: Return $(record, \sigma_{S_j})$

Algorithm 3 S_j.GetStatus ()

1: Return $S_j.Status$

$N_{p,l}$ (line 1). If not, it returns $\perp$ (line 2). Otherwise, the server retrieves the corresponding record from DB (lines 4-5). Note that as we discussed in the $S_j.Write$ algorithm, depending on the type of requested node some or all of the entries $(N_{p,l}, post, \sigma)$ might be empty. If the requested node is a tree digest, then the server generates a signature over $N_{p,l}$ (lines 6-10). Finally, the server sends the *record* and the signature (if any) to the user (line 11).

3. S_j.***GetStatus***(): The server sends its *Status* to the user (Algorithm 3).

User-side Protocols:

Create *object*: This protocol aims to initialize the share *object* by the insertion of its first post and then to communicate the first tree digest TD_1 with the authorized users. The content of the first post must be uniquely representative for the *object* e.g., the name of the group page together with its creation date. As such, given the first tree digest of the shared *object*, users can distinguish between different shared *object* s (e.g., different group pages). We assume one of the authorized users U_i e.g., the owner of a wall or the admin of a Facebook-like group page will run this protocol. The input of the user to this protocol is a *post*. The user contacts the server $S_{F(1,1)}$ who is responsible for the first post and calls its $Write$ function for $(U_i, (1, 1), (H(post||1), post, Sign_{U_i}(H(post||1))$. As the result, user receives a signature $\sigma_{S_{F(1,1)}}$ over the inserted value $TD_1 = H(post||1)$ from the server. Note that, the hash value $H(post||1)$ corresponds to the $N_{1,1} = TD_1$.

The admin then communicates TD_1 with all the authorized users in $FList$. Each user initializes a local variable *Status* for the shared *object* which is the tuple of the following format $Status = (v, TD_v, \sigma)$ where v reflects the last version of shred *object* seen by the user, TD_v is the corresponding tree digest and σ is a server-side signature of the TD_v. Each user sets the *Status* variable to $(1, TD_1, \sigma_{S_{F(1,1)}})$.

Update Status: This protocol, as demonstrated in Figure 10, is run between a user and the N servers through which the user aims to find the index of the latest post uploaded on the *object*, to fetch the corresponding tree digest, and check its consistency against her local Status variable. As such, the user collects the *Status* value of all the servers via their *GetStatus* function call (line 1). The largest *Status* value indicates the latest post index i.e., *last*. If the label of last nodes seen by the servers differ in more than N (N is the total number of servers), then inconsistency is detected (lines 2-3). This is because servers get to serve nodes in a circular manner, hence, the difference between the labels of the nodes seen by the servers can be at most N.

Next, the user must check whether her last seen tree digest TD_v (stored in her *Status* variable) (line 4) is consistent with the given tree digest TD_{last}. To do so, the user identifies the nodes holding the path of an incremental proof between tree digest TD_v and TD_{last} (line 5) and contacts the corresponding storage servers (lines 6-7). Next, she authenticates the retrieved signatures of leaf nodes and tree digests (lines 6-12). If the authentication succeeds, then the user checks whether the fetched proof is a valid incremental proof between TD_v and TD_{last} (line 13). If verified, the user updates her *Status* value to $(last, TD_{last}, \sigma_{S_{F(last,\lceil log(last)\rceil+1)}})$ (line 14). Note that $\sigma_{S_{F(last,\lceil log(last)\rceil+1)}}$ is fetched as a part of $proof$.

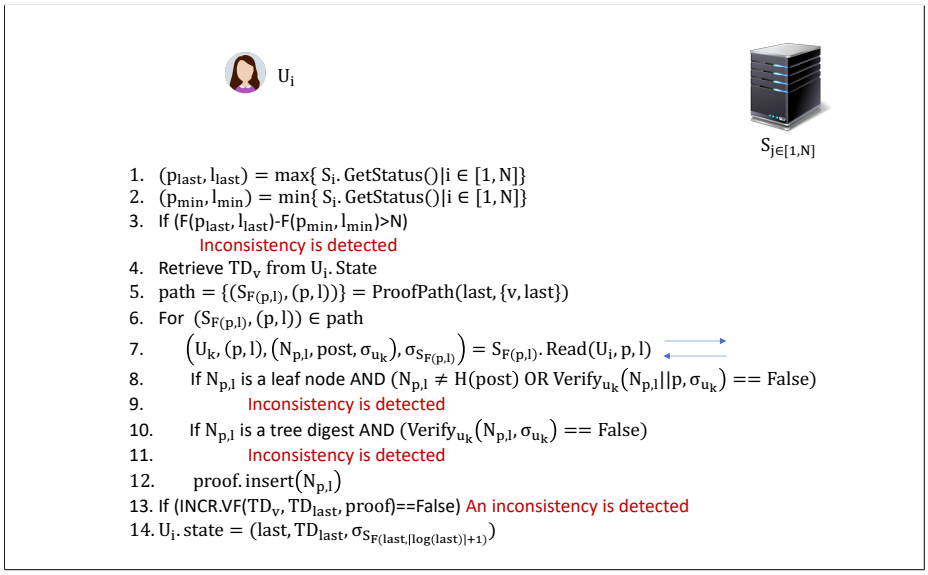

Fig. 10. Update Status protocol. The arrows indicate users interaction with servers.

Read: During the *Read* protocol as shown in Figure 11, the user reads a certain range $R = [x, y]$ of posts i.e., $post_x, \cdots, post_y$ of the shared *object* (line 1).

For this, the user first runs the *UpdateStatus* protocol and updates her *Status* variable (line 2). Next, she specifies the storage servers holding the nodes on the membership proof path of $post_{i \in R}$ (line 3). She contacts the servers and fetches the required data (lines 4-11). The fetched tree digest and leaf nodes should be appropriately signed by the issuing users (lines 6-9). Once the signatures are verified, the user verifies the correctness of membership proofs of the posts against TD_{last} (line 12). If the result of membership proof is false then a view inconsistency is detected (line 13).

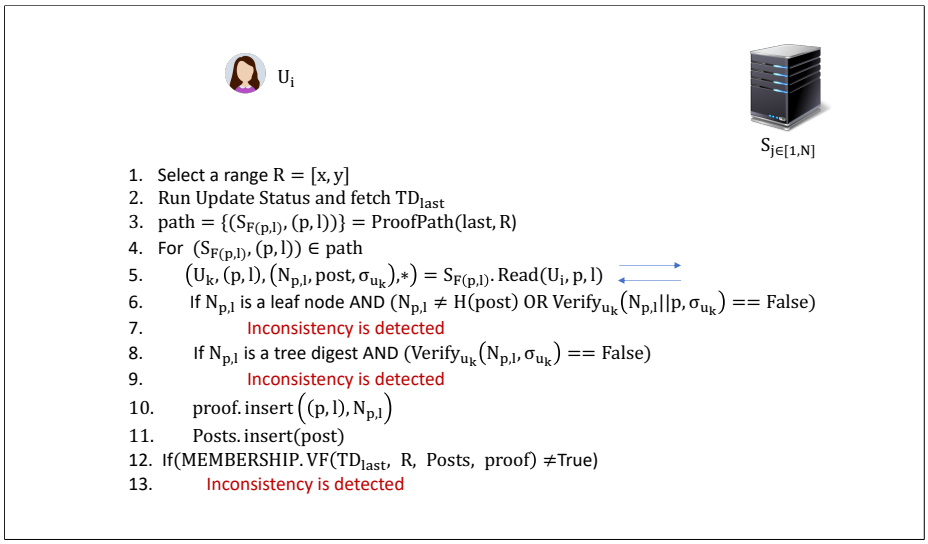

Fig. 11. *Read* protocol. The arrows indicate the user's interaction with servers.

Write: In this protocol, illustrated in Figure 12, a user interacts with N servers to insert her post to the *object*. User initially runs *Update Status* protocol to fetch TD_{last} corresponding to the latest version of the *object* (line 1).

The user U_i crafts the content of her post (line 2) and signs it (line 3). She identifies the nodes on the insertion path of her post (line 5) and then fetches the values from the corresponding storage servers (lines 6-10). Next, she recomputes the hash values of intermediate nodes on the insertion path of her post (line 11). She submits the hash values to the corresponding servers by invoking their *Write* function (lines 12-16). For the leaf node, the user submits the hash value $N_{last+1,1}$ together with the content of the post *post* and a signature σ (line 13). For the full node, the value of the node is sent to the corresponding server (line 14) whereas the storage server of the temporary node just gets informed about the insertion of the new post (line 15). If any of the servers respond by reject, then inconsistency is detected (line 17). The storage provider of the tree

digest i.e., N_{p_c,l_C} returns a signature (line 18). The user authenticates the given signature and updates her Status accordingly (lines 16-17).

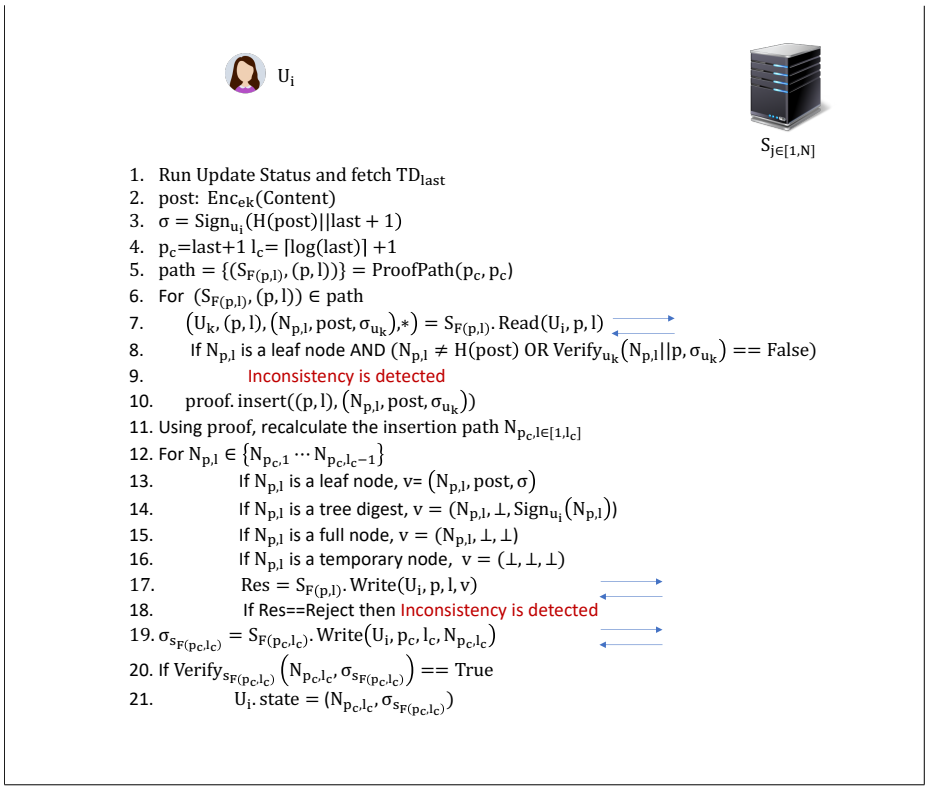

Fig. 12. *Write* protocol. The arrows indicate the user's interaction with servers.

Audit: As we discussed before, each user is responsible to ensure that her post is correctly inserted to the *object* and is visible to all the other users. As such, every write operation of i^{th} post must be followed by a call to the *UpdateStatus* at $i+q^{th}$ version of the *object*. That is, once the user uploads a post to the shared *object*, she must check the status of the shared *object* when q more posts are uploaded on top of her post. If the execution of *UpdateStatus* at $i+q^{th}$ version of *object* concludes successfully, then the user ensures that her post is consistently visible to all the other users. Otherwise, an inconsistency is detected.

Audit Threshold: The value of q is a function of *object* version i (the index of inserted post) and the number of servers N as shown in Equation 7. We refer to q as *audit threshold* i.e., a threshold for the number of posts that a user needs to wait to be inserted on top of her post to ensure that her post is consistently visible to all the users.

$$Q(i, N) = min(q) \text{ s.t. } \sum_{j=0:q} \lceil log(i+j) \rceil + 1 \geq N \tag{7}$$

As a concrete example, assume a system with 8 servers i.e., $N = 8$. A user who inserts the second post i.e., $i = 2$ shall execute *UpdateStatus* after the insertion of $Q(2, 8) = 2$ more posts on the *object* i.e., at the 4^{th} version of the *object*. If the *Update Status* protocol does not end successfully, then there is a view inconsistency e.g., servers attempted to drop her post or replace with another post.

We give some intuitions into why auditing the *object* after $Q(i, N)$ posts will result in achieving q-detectable consistency. In section 5, we will provide a formal security definition for a q-detectable consistent system together with solid proof as to how Integrita satisfies q-detectable consistency relying on our proposed auditing strategy.

For the insertion of each post i, servers that are located on the insertion path will be informed about the insertion of that new post regardless of the type of nodes they are responsible for (see Figure 12). Recall that we assume at least one of the servers is honest. We use the term of frozen post for a post whose part of the insertion path gets to be served by the honest server by frozen post. It is named frozen since due to the presence of the honest server, no other post with the same index as the frozen post will exist; the honest server will not accept the insertion of two posts with the same index (as indicated in line 2 of Algorithm 1). This implies that for the frozen post with the index of f, there would be only one tree digest TD_f in the system which represents a unique history (sequence of posts) of the *object*. We call a tree digest corresponding to a frozen post as frozen tree digest. All the other tree digest TD_j created as the result of further write operations $j > f$ will comply with the history that TD_f represents (this is due to the incremental proof check in the *Step 11 of the Update Status* protocol). Thus, if a post i where $i < f$ belongs to the sequence of posts that a frozen tree digest TD_f represents then it will certainly belong to all the future versions of the *object* (again due to the incremental proof check in the *Step 11 of the Update Status* protocol). Thus, to ensure view consistency, the user needs to perform a consistency check between the tree digest at the time of insertion of her post and the very next frozen tree digest. To determine the index of the next frozen tree digest, we need to know the index of the honest server. However, there is no presumption about which server will act honestly. As such, after insertion of each post i the user shall wait for q many posts to be inserted as the result of which all the servers get contacted at least once. As the storage of nodes is assigned to the servers under a circular pattern, if the sum of the length of the insertion path of the next q posts exceeds N, it means that all the N servers, including the honest server whose index is unknown, are contacted at least once. Equation 7 calculates q i.e., the total number of posts (inserted after i^{th} post) whose insertion paths lengths on aggregate exceeds N. Recall that the number of nodes located on the insertion path of j^{th} post is $\lceil log(j) \rceil + 1$ which means

$\lceil log(j) \rceil + 1$ distinct servers get contacted as the result of insertion of the j^{th} post.

Analysis of Audit Threshold Figure 13 shows the audit threshold computed based on function $Q(i, N)$ under different number of servers N and post number i. The audit threshold for a particular post number will increase with the number of servers e.g., the threshold audit for post number 65 for $N = 8, 16, 24$, and 32 are $0, 1, 2$ and 3 respectively.

After a certain version of *object*, every inserted post is a frozen one since all the servers get contacted as the insertion of each post. Indeed, the *object* enters into its strong consistent version where no fork can happen in users' views. We call that version of the *object* as *transition point*. For a given N, its transition point is computed as given in Equation 8.

$$TP(N) = 2^{N-2} + 1 \tag{8}$$

For example, with $N = 8$, strong consistency starts at version 65 whereas with $N = 16$ the transition point is 16385. Thus the higher the number of servers the later the *object* enters its strong consistency version. The transition points of the different numbers of servers (1-20) are illustrated in Figure 14.

4 Complexity and Performance

In this section, we analyze the asymptotic performance of Integrita with respect to the storage overhead (section 4.1), communication round and communication complexity (section 4.2) for both servers and users.

4.1 Storage Overhead

User: Users have to store constant amount of data as for their Status variable.
Server: Servers are responsible to store the object's posts and its associated history tree i.e., the full nodes, the tree digests, and the leaves.

$$hSize \cdot |L| + hSize \cdot |F| + hSize \cdot |T - F| + SSize \cdot |L| + SSize \cdot |T| \tag{9}$$

An object with M posts consists of M leaves, and M full nodes and M tree digests. However, out of M tree digests, some of them overlap with the full nodes hence are already saved in the system. Indeed, out of M tree digests (for M posts), $log(M)$ many of them associated with post indices 2^0, 2^1, 2^2,..., $2^{log(M)}$) are full nodes, hence, servers the number of tree digests to be stored by the servers will be $M - log(M)$ (rather than M). Additionally, every leaf node is attached to a user-side signature, thus, M signatures shall be maintained by the servers. Also, for an object with M posts, $O(N)$ many signed tree digests are stored by the servers which result in $O(N)$ signatures.

As such, the total amount of storage spent by the N servers is given in Equation 10 which is of $O(M)$.

$$hSize \cdot (\underbrace{M + M}_{\text{leaves and full nodes}} + \overbrace{M - log(M)}^{\text{tree digests that are not full nodes}}) + SSize \cdot (M + N) \tag{10}$$

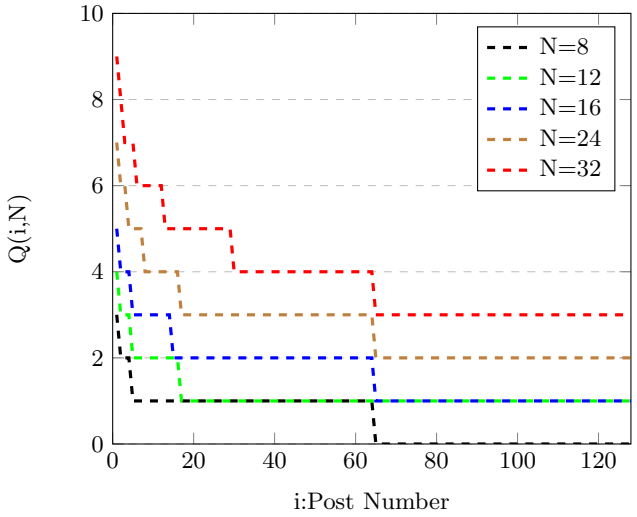

Fig. 13. The Audit Threshold for various number of servers each demonstrated by a different diagram. The x axis represents the index of post whereas the y axis shows the audit threshold computed based on function $Q(i, N)$ given in Equation 7.

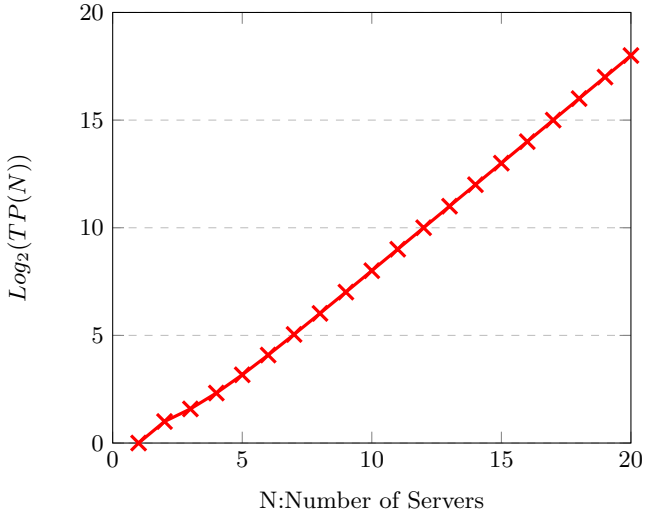

Fig. 14. The Transition point for various number of servers as defined in Equation 8. The x axis represents the number of servers whereas the y axis shows the the logarithm of the transition point.

Design	Asymptotic Overhead	Concrete total overhead	Concrete overhead per server	Consistency level
Centralized	$hSize \cdot (2 \cdot M) + SSize \cdot M$	124TB	124TB	Fork-consistency
Replication-based	$N \cdot [hSize \cdot (2 \cdot M) + SSize \cdot M]$	2488TB	2488TB	Strong consistency
Integrita	$hSize \cdot (3M - Log(M)) + SSize \cdot (M + N)$	143TB	7.1TB	q-detectable consistency

Table 1. Storage overhead of Integrita vs related work. N: number of servers. M: number of posts on the object. $SSize$: The size of each signature in bit length. $hSize$: the bit-length of hash output.

Related Work: In a replication-based solution, one needs to copy the history tree (of $2M$ nodes) as well as the signed leaves of the history tree (i.e., M signatures) over N servers. Thus, the storage overhead for such design would be $N \cdot [hSize \cdot (2 \cdot M) + SSize \cdot M]$. In the centralized systems, the server stores the history tree (With $2 \cdot M$ nodes) together with its user-side authenticated leaves (i.e., M signatures) which results in $hSize \cdot (2 \cdot M) + SSize \cdot M$ overhead.

Table 1 summarizes the comparison of the storage overhead of Integrita with the centralized and the replication-based solutions. The concrete overhead is measured for annual storage consumption of a social network like Facebook, for walls of its 2.41 billion monthly active users[3] each wall containing 1241 posts (per year) [4]. We deploy SHA-3 as the hash function with a 512-bit output length and RSA signature scheme with a 2048-bit signature length. The number of servers N is set to 20.

4.2 Round Complexity and Communication Complexity

In this section, we analyze the round complexity as well as the communication complexity i.e., the number of bits communicated between parties during the protocol. We consider each communication round to be a sent and a receive operation. If multiple rounds can be done concurrently (they are independent), then we count them as one round. The results are summarized in Table 2.

- *Update Status:* In the first round, the user contacts all the N servers to get their latest Status. Then she performs a consistency proof check between her local state variable and the latest state of the object. To fetch values of the incremental proof path, the user contacts with N servers (at most) and downloads the necessary values concurrently. Thus, the overall round complexity of *Update Status* is 2. Likewise, as the result of *Update Status*, each server may get contacted twice, once to share its latest status and the second time when the server may be located on the proof path.
  The communication complexity of *Update Status* is to download the signed state of servers S_i for $i \in [1, N]$ as well as fetching the incremental proof. The former requires $SSzie \cdot N$ data transfer whereas the latter involves the transmission of at most $2 \cdot log(M)$ hash values (M is the number of object's

[3] https://www.businessinsider.com/facebook-grew-monthly-average-users-in-q1-2019-4

[4] https://blog.wishpond.com/post/115675435109/40-up-to-date-facebook-facts-and-stats

Entity\Overhead	Update Status	Read	Write
User	2	1	2
Servers	2	1	2

(a) Integrita Communication Complexity

Entity\Overhead	Update Status	Read	Write
User	$SSize \cdot (N+3) + hSize \cdot 2 \cdot log(M)$	$hSize \cdot (2.log(M) + R) + SSize \cdot R$	$hSize \cdot 2 \cdot log(M)$
Servers	$SSize + hSzie \cdot \frac{2 \cdot log(M)}{N}$	$\frac{hSize \cdot (2.log(M)+R)+SSize \cdot R}{N}$	$\frac{hSize \cdot 2 \cdot log(M)}{N}$

(b) Integrita Communication Complexity.

Table 2. Communication complexity. N: number of servers. M: number of posts on the object. $SSize$: The size of each signature in bit length. $hSize$: the bit-length of hash output. R: number of consequitive operations to be read from the object.

posts). The user additionally downloads the user authenticated version of the last post's tree digest and leaf node as well as the server-signed version of the tree digest which adds 3 more $SSize$ to the amount of transferred data. As such, the communication complexity at the user side is at most $SSize \cdot (N+3) + hSize \cdot 2.log(M)$. The average communication complexity for each server is $\frac{SSize \cdot (N+3) + hSize \cdot 2.log(M)}{N} \approx SSize + \frac{hSize \cdot (2 \cdot log(M))}{N}$

- *Read:* Let the object contains M posts, and a user wants to read a consecutive range $R = [i, j]$ of posts. She has to fetch the membership proof paths of all the posts from the corresponding servers. The user can connect to all the servers simultaneously, hence she can perform read in 1 round of communication. As a result, each server at most gets contacted also once which results in communication complexity of 1 for each server.

  The user downloads R many leaf nodes with their user-side signatures which requires $R \cdot (hSize + SSize)$ data transmission. The proof path includes at most $2 \cdot log(M)$ hash values. On aggregate, user communicates $hSize \cdot (2.log(M) + R) + SSize \cdot R$ bit data. Consequently, the average data transfer on each server is $\frac{hSize \cdot (2.log(M)+R)+SSize \cdot R)}{N}$.
- *Write:* To insert a post to the object, the user needs to fetch the nodes on the insertion path of her post. This can be handled in a 1 round of communication with concurrent connections to the servers. Next, the user recomputes the values for the nodes on the insertion path of her post and upload the new values to the servers. This also counts as a 1 round of communication. Thus, in total user performs the write operation in 2 rounds of communications. Subsequently, servers may get contacted for at most 2 rounds.

  From the communication complexity perspective, downloading the insertion path of the current post requires to download at most $2 \cdot log(M)$ hash values. As such, the user-side communication complexity would lead to $hSize \cdot 2 \cdot log(M)$ bits. The data transfer at the server-side shall be $\frac{hSize \cdot 2 \cdot log(M)}{N}$.

5 Security

5.1 q-Detectable Consistency and Inconsistency Interval

In a q-detectable consistent system, views of users toward the i^{th} version of a shared
object is guaranteed to be consistent expect for the last δ posts i.e., $post_{i-\delta}, ..., post_i$. We use the term *inconsistency interval* to refer to the range of the posts i.e., $[i-\delta, i]$ where the inconsistency is allowed. The views of users for any history of
object preceding $i-\delta$ version of the
object is guaranteed to be the same. In Integrita, δ is a function of
object version i and the number of servers N and its value is computed using function $\Delta(i, N)$ given in Equation 11.

$$\Delta(i, N) = max(q \in [0, i]) \text{ s.t. } \sum_{j=0:q} \lceil log(i-j) \rceil + 1 \geq N \tag{11}$$

As a concrete example, assume a system with 8 servers i.e., $N = 8$ and two users looking at the 5^{th} version of the
object, the inconsistency interval is 2 ($\Delta(5, 8) = 2$) that is the view consistency holds for all the posts except the 4^{th} and the 5^{th} post. As such, the following two views $View_5 = \{post_1, post_2, post_3, post_4, post_5\}$ and $View'_5 = \{post_1, post_2, post_3, post'_4, post'_5\}$ are q-consistent since the consistency holds for all the posts out of the inconsistency interval. However, the following two views $View_5 = \{post_1, post'_2, post_3, post_4, post_5\}$ and $View'_5 = \{post_1, post'_2, post_3, post'_4, post'_5\}$ do not satisfy q-consistency because there is an inconsistency at the second post which is out of the inconsistency interval.

To capture the notion of q-consistency, we define the following game to be played between an adversary $\mathcal{A}$ and a challenger $Chal$. The adversary shall control $N-1$ servers whereas the $Chal$ gets to play for authorized users U_i $i \in FList$ and the honest server. We write F to indicate the indices of corrupted servers and S_h to be the honest server. The adversary can dictate the read and write operations to be done by particular users. However, it does not have control over the Audit protocol execution. The challenge for the adversary is to make two users U and U' accept two q-inconsistent views of the i^{th} version of the *object* i.e., there is at least one index $j \notin [i - \Delta(i, N), i]$ for which U and U' read $post_j$ and $post'_j$ as the j^{th} post such that $post_j \neq post'_j$.

q-Detectable Consistency Experiment q-Det-Consistency(1^λ)

1. The challenger gives the security parameter 1^λ to the adversary. The adversary communicates a set of signature verification keys $\{vk_{S_i}\}_{i \in F}$ for the servers under its control to the challenger. The challenger runs the signature key generation algorithm for the honest server and hands the vk_{S_h} to the adversary. Also, the challenger generates the signature key pairs for the users $U_1, ..., U_T$ and outputs $FList = \{(U_1, vk_{U_1}), \cdots, (U_T, vk_{U_T})\}$ to the adversary.
2. The adversary specifies a user U_j to create the shared *object* $\mathcal{D}$ through the invocation of *Create object* protocol.
   Steps 3 and 4 can be repeated polynomial times by the adversary.
3. The adversary specifies a user U_j to $Write$ a *post* on the *object* $\mathcal{D}$. *Chal* runs the $Write$ protocol accordingly. Note that after each write operation, the challenger shall act upon the Audit protocol.
4. The adversary specifies a range $R = [l, r]$ to be read by a particular user U_i. *Chal* runs the *Read* protocol accordingly.
5. The adversary specifies two users U and U', a version number j and a post index i^* where $i^* < j - \Delta(j, N)$. The challenger runs *Read* protocol for j^{th} post on behalf of U and U'. $\mathcal{A}$ wins if *Read* protocol ends successfully for U and U' such that the tree digest in the Status variable of U and U' both have index j and $post_{i^*}$ and $post'_{i^*}$ be the posts the U and U' read, respectively s.t. $post_{i^*} \neq post'_{i^*}$.

Definition 1. *A storage system provides q-detectable consistency for a shared object if the success probability of adversary in q-Det-Consistency(1^λ) experiment is negligible in the security parameter λ.*

Theorem 1. *If the deployed signature scheme is existentially unforgeable under adaptive chosen message attack and the hash function is secure then Integrita satisfies q-detectable consistency.*

Proof Overview: If A wins i.e., U and U' read two different post $post_{i^*} \neq post'_{i^*}$ where $i^* < j - \Delta(j, N)$ this implies that there is a fork in the system where users are split into two groups depending on whether they are shown $post_{i^*}$ or $post'_{i^*}$. For both forks to successfully continue till the j^{th} version of the object, each fork should have a successful chain of write operations from i^{*th} to j^{th} version of the object. However, for the frozen $post_k$ where $i^* < k < i^* + Q(i^*, N) < j$, the honest server accepts only one write operation which results in one valid tree digest hence only one fork will get to grow. For the other fork (namely the second fork) to grow, the corrupted servers need to bypass the honest server. As such, the corrupted servers need to convince the users of the second fork that the last post on the object has an index higher than k so that they won't attempt insertion of the k^{th} post. However, this would only happen if the corrupted servers can generate an authenticated post and tree digest on behalf of an authorized user

from the second fork. Thus, if B can guess for which authorized user this forgery takes place, B will exploit this forgery and breaks the unforgeability of the underlying signature scheme.
Proof: We base our proof over the following lemma that is due to [7].

Lemma 1. *If there is a valid incremental proof between two tree digests TD_i and TD_j, then for every operation $post_k$ where $k < i$ for which there is a valid membership proof, s.t. $True \leftarrow MEMBERSHIP.VF(k, TD_i, post_k, proof))$, and $post'_k$ s.t. there is a $proof'$ for which $True \leftarrow MEMBERSHIP.VF(k, TD_j, post'_k, proof))$ then $post_k$ must be equal to $post'_k$. Namely, if two tree digests are consistent then they both represent the same sequence of operations for their shared past [7].*

Proof: If there exists an adversary $\mathcal{A}$ who wins q-Det-Consistency(1^λ) with non-negligible probability ϵ then we construct a simulator B who breaks the underlying signature scheme. The internal code of B is given below. B is given the security parameter 1^λ as well as a signature verification key vk' from the signature scheme challenger.

1. The challenger gives the security parameter 1^λ to the adversary. The adversary communicates a set of signature verification keys for the corrupted servers $\{vk_{S_i}\}_{i \in F}$ to the challenger. The challenger runs the signature key generation algorithm for the honest server and hands the vk_{S_h} to the adversary. B selects a random value $\beta \leftarrow [1, T]$. B sets the signature verification key of U_β to vk' and for the rest of users generates the signature key pairs as normal. B sends $FList = \{(U_1, vk_1), \cdots, (U_\beta, vk'), \cdots, (U_T, vk_T)\}$ to the adversary.
2. The adversary specifies a user U_j to create the shared *object* $\mathcal{D}$ through the invocation of *Create object* protocol. If $j = \beta$, then to generate required signatures, B queries the signing oracle of the outside challenger and stores the set of queried messages and signatures in set $QSign$. Otherwise, B acts as in *Create object* protocol.
3. The adversary specifies a user U_j to write a post on the *object* $\mathcal{D}$. B runs the $Write$ protocol accordingly.
   First, B runs the Update Status and fetch the latest tree digest TD_{last}. As the result of running Update Status, B obtains $proof = \{(N_{p,l}, post, \sigma_{U_k})\}$ for some p and l. If there exists a tree digest $N_{x,y} \in proof$ (or a leaf node) signed by U_β as σ_{U_β} s.t. $N_{x,y} \notin QSign$ (or $N_{x,y}||x \notin QSign$) then B outputs ($N_{x,y}$, σ_{U_β}) (or ($N_{x,y}||x$,σ_{U_β})) to the outside challenger.
   B fetches required nodes for the insertion of the new post as $proof = \{(N_{p,l}, post, \sigma_{U_k})\}$. If there exists a tree digest $N_{x,y} \in proof$ (or a leaf node) signed by U_β as σ_{U_β} s.t. $N_{x,y} \notin QSign$ (or $N_{x,y}||x \notin QSign$) then B outputs ($N_{x,y}$, σ_{U_β}) (or ($N_{x,y}||x$,σ_{U_β})) to the outside challenger.
   B recalculates the nodes on the insertion path of her post as well as the tree digest. B signs the leaf node i.e., $H(post)||last + 1$ and the tree digest TD_{last+1} using the U_j signature key. If $U_j == U_\beta$ then B queries the signing

oracle of the outside challenger and inserts the queried message and the obtained signature to $QSign$.
If an inconsistency is detected as the result of $Write$ protocol, B immediately aborts.
Note that after each write operation, B shall act upon the audit protocol i.e., B runs the Update Status protocol at the $last + Q(last, N)$ version of the
object. As the result of running Update Status B obtains $proof = \{(N_{p,l}, post, \sigma_{U_k})\}$. If there exists a tree digest $N_{x,y} \in proof$ (or a leaf node) signed by U_β as σ_{U_β} s.t. $N_{x,y} \notin QSign$ (or $N_{x,y}||x \notin QSign$) then B outputs ($N_{x,y}$, σ_{U_β}) (or ($N_{x,y}||x$,σ_{U_β})) to the outside challenger. If any inconsistency is detected as the result of Update Status, then B aborts.

4. The adversary specifies a range $R = [l, r]$ to be read by a particular user U_i. B runs the *Read* protocol accordingly. If an inconsistency is detected as the result of *Read* protocol, then B aborts. Otherwise, during the execution of *Read* protocol, B obtains $proof = \{(N_{p,l}, post, \sigma_{U_k})\}$. If there exists a tree digest $N_{x,y} \in proof$ (or a leaf node) signed by U_β as σ_{U_β} s.t. $N_{x,y} \notin QSign$ (or $N_{x,y}||x \notin QSign$) then B outputs ($N_{x,y}$, σ_{U_β}) (or ($N_{x,y}||x$,σ_{U_β})) to the outside challenger.
5. The adversary specifies two users U and U', a version number j and a post index i^* where $i^* < j - \Delta(j, N)$. B runs *Read* protocol for U and U' separately. B acts identically to the step 4 to run the *Read* protocol. Let TD_j and TD'_j indicate the Status variable of U and U' after the *Read* protocol execution . Also let $post_{i^*}$ and $post'_{i^*}$ indicate the read posts for U and U'. If $post_{i^*} \neq post'_{i^*}$ and *Read* protocol ends without detecting any inconsistency for both U and U' then B will find a signature forgery as we discuss below.

Note that the inconsistency between $post_{i^*}$ and $post'_{i^*}$ means that there will be two different tree digests TD_{i^*} (with $post_{i^*}$ as its i^{*th} post) and TD'_{i^*} (with $post'_{i^*}$ as its i^{*th} post). As such, from version i^* onward, users will be divided into two groups G and G' depending on whether they are shown $post_{i^*}$ (TD_{i^*}) or $post'_{i^*}$ (TD'_{i^*}). More precisely, a group G of users whose further Status variables i.e., TD_f where $f \geq i^*$ are consistent with TD_{i^*} i.e., $TD_{i^*} \rightarrow TD_f$ and the other group G' whose further Status variables i.e., TD'_f where $f \geq i^*$ are consistent with TD'_{i^*} i.e., $TD'_{i^*} \rightarrow TD'_f$.

Recall that every read and write operation requires the user to run the Update Status protocol, and to perform an incremental proof check between local Status variable and the latest state of the system i.e., TD_{last}. Since users are divided in two groups G and G', there will be two separate chains of posts (after i^{*th} post) generated by group G and G' i.e., $post_i$ $i \in [i^*, j]$ uploaded by group G and $post'_i$ $i \in [i^*, j]$ performed by users of group G'. Let assume $k \in [i^*, i^* + Q(i, N)]$ be the index of the next frozen node (the honest server is the storage server of one of the nodes on the insertion path of post k). Assume that a user from a group G attempts the insertion of $post_k$ earlier than a user from a group G'. Since the honest server appears on the insertion path of $post_k$, it gets informed about the inclusion of k^{th} post and update its $Status$ accordingly. When a user from the

group G' holding a state variable TD'_i wants to insert $post'_k$, it first runs the Update Status to fetch the latest version of the
object and perform consistency check between TD'_i and the current version of the
object. During the status update protocol, the adversary may try to act dishonestly which we discuss next.

1. The adversary may attempt to send an incorrect Status value to the user and make her accept a lower version $< k$ of the
   object. However, due to the presence of the honest server (who has witnessed the insertion of $post_k$), the adversary does not succeed as the honest server will communicate its intact state value i.e., k with the user.
2. The adversary may attempt sending a Status value x where $x \geq k$ for which the adversary also needs to come up with a valid tree digest TD'_x where $TD'_x \implies TD'_i$ (TD'_i is the status of user while inserting $post'_k$) in order to pass the Update Status protocol successfully. To come up with a valid TD'_x, the adversary has the following choices:
   (a) The adversary may use the tree digest TD_x that is signed and generated by one of the members of the group G. However, any tree digest TD_x generated by a member of group G will be consistent with TD_{i^*} but not with TD'_{i^*} i.e., $TD_{i^*} \not\Longrightarrow TD_x$. This means that there will be no valid incremental proof between TD'_{i^*} and TD_x. Thus this choice is absolute.
   (b) The other choice for the adversary is to generate a $post'_x$ and forge a signature on $H(post'_x||x)$ (the leaf node) on behalf of an authorized user.
   (c) The adversary uses $post_x$ generated by one of the members of G and computes the tree digest TD'_x accordingly. $\mathcal{A}$ also needs to generate a valid signature over TD'_x from one of the authorized users.

   This means that for a member of group G' to accept that the latest version of
   object is $x \geq k$ and successfully pass the Update Status protocol, the adversary needs to forge a signature on behalf of an authorized user U'' either on the leaf node $H(post'_x||x)$ or the tree digest TD'_x. Thus, B shall figure out this forgery while fetching the incremental proof on behalf of a member of the group G'.

B can win the signature game if the forgery of the adversary is from U_β. Recall that the probability of $\mathcal{A}$ winning the q-Det-Consistency(1^λ) is $\epsilon(\lambda)$ and the total number of users i.e., T is a polynomial $poly(\lambda)$. Thus, we have

$$\begin{aligned} Pr[B \text{ breaks the signature}] &= Pr[\text{q-Det-Consistency}(1^\lambda) \ = 1 \text{ AND } U'' = U_\beta] \\ &= Pr[\text{q-Det-Consistency}(1^\lambda) \ = 1 | U'' = U_\beta] \cdot Pr[U'' = U_\beta] \\ &= \epsilon(\lambda) \cdot \frac{1}{T} \\ &= \epsilon(\lambda) \cdot \frac{1}{poly(\lambda)} \end{aligned} \tag{12}$$

if $\epsilon(\lambda)$ is non-negligible, then B also breaks the signature scheme with non-negligible probability. This concludes the proof. ■

6 Related Works

The concern of view consistency similarly is investigated in the context of centralized OSN, peer-to-peer (p2p) OSN, cloud storage, Byzantine fault-tolerant protocols, and authenticated data structures. In the following, we elaborate on these proposals and compare them with Integrita.

Centralized OSN: In centralized architecture of OSNs, frientegrity [9] and SPORC [10] address the view consistency by achieving fork-consistency on a shared data. In the fork*-consistent system, while a corrupted provider is able to fork the users into disjoint sets, he is forced to serve each set with a consistent view of operations performed by the users of the same set. This is enabled since users embed their views of the object history in each post they insert. Thus, as soon as the server forks view of two users, he cannot show their operations to each other without risking detection. Users can also detect the inconsistency of their views by exchanging them out of the band. The main shortcoming of fork*-consistent systems is that the server's equivocation remains undetected till users happen to contact out of the band. Thus, to ensure view consistency, users must regularly communicate their views of the shared object. This approach would not be practical. For example, on Facebook, each user has on the average 338 friends [5] which would all have access to her wall as a shared object. Hence, each user needs to communicate with almost 338*338= 114244 other users to monitor the view consistency of her wall and her friends' walls.

Peer-to-Peer OSN: In a p2p OSN, there is no central server running the system and instead, the individual users called peers contribute a part of their computation and storage power to the system. The social networking services are enabled in a distributed manner relying on shared resources. As such, the storage of users' data is also distributed among the existing peers. The view consistency in p2p OSNs is usually addressed through replication or by leveraging users' trust. In the latter case, the object owner (e.g., owner of a wall) stores and serves her wall by herself or replicates on some trusted peers like her friends. Subsequently, the view consistency is guaranteed due to the trustworthiness of storage peers [26, 14, 18, 3, 27, 29]. However, if the storage responsibility is spread over the p2p network and the storing peers are untrusted, then view consistency is met through replication [22]. In particular, suppose f as the fraction of potential dishonest peers, the object (or some units of the object like a post) should be replicated on $f+1$ peers to ensure that at least one honest peer is among the replicas. Each requester reads each post from all the $f+1$ replicas and identifies the latest content (e.g., using a version number). However, such a solution results in storage overhead and communication complexity which grow linearly by f. Other studies in the context of p2p OSNs also utilize replication but for the sake of data availability [25, 21, 29]. Namely, the storing nodes are supposed to be trusted and always serve the intact contents when available.

Byzantine Fault-Tolerant Protocols and Cloud Storages: In BFT protocols, a service is to be given to a set of clients while the execution of

[5] https://www.brandwatch.com/blog/facebook-statistics/

the service concerning the sequence of requests appears identical to all clients (and this sequence preserves the temporal order of non-concurrent operations). Enabling consistency, BFT protocols also seek the replication-based solution [17, 6] where they deploy several servers each keeping a replica of the state of the intended service. Byzantine fault-tolerant systems behave correctly when no more than f out of $3f + 1$ replicas fail [17].

Similar to the centralized OSNs, the best level of view consistency in the context of cloud storage is fork-consistency [5, 4, 19, 2, 30] which is due to the presence of a corrupted service provider and non-communicating users. Addressing the fork issue, cloud storage platforms utilize replication over multiple servers [15, 17].

Authenticated Data Structure: In the Authenticated data structures, a data owner outsources her data to multiple untrusted repositories. The outsourced data is modeled by a data structure that enables performing queries on the data in a verifiable and authenticated manner. Repositories, on behalf of the data owner, are responsible to answer queries of users on the data structure and hand them with a proof of the validity of the answer. The same data structure is replicated over all the repositories and repositories need to keep themselves updated with the data owner in the case of update [11, 12, 24, 28, 23]. As such, one can assume view consistency of ADSs is guaranteed through replication which is not storage efficient.

7 Conclusion

In Integrita, we address the view consistency issue in a collaborative data-sharing environment like Facebook group pages. The shared data called a shared object is comprised of a sequence of posts which can be generated by any of the authorized users. The view consistency concerns that all the authorized users are shown the same set of posts and with the intact order. In Integrita, we introduce a new level of consistency called *q-detectable consistency* where any inconsistency between users view cannot remain undetected for more than q posts. Integrita preserves q-detectable consistency as long as one server does not collude with the rest of the servers. In Integrita, q is a function of the number of posts uploaded on the shared object as well as the number of servers. Integrita outperforms the state of the art in two major directions. First, unlike the replication-based solutions, Integrita operates only on one instance of the shared object that is maintained collaboratively by all the servers. As such, Integrita saves 2344 Terabyte of storage annually for an OSN with 2.3 billion users and running on the federation of 20 servers (i.e., service providers). We enable this by trading the strong consistency with the q-detectable consistency. Second, in contrast to the centralized counterparts in which the inconsistency detection relies on the users' direct communication, Integrita detects any fork in the users' views regardless of users direct communication. Nevertheless, distributing the storage of shared data among multiple servers not only does not degrade the performance of our design compared to the centralized architecture, but also our complexity analysis

shows that Integrita performs identically to the centralized architecture concerning the communication and computation both at the user and the server-side. Also, Integrita reduces the storage overhead per server by a factor of N (N is the number of servers). Additionally, Integrita does not rely on cross-server communication in resolving users' read and write requests.

As our future work, we would extend Integrita to support view-consistency in the adversarial model where users may also be corrupted.

Acknowledgements

We acknowledge the support of the Turkish Academy of Sciences, the Royal Society of UK Newton Advanced Fellowship NA140464, and European Union COST Action IC1306.

References

1. Bethencourt, J., Sahai, A., and Waters, B. Ciphertext-policy attribute-based encryption. In *IEEE symposium on security and privacy (SP'07)* (2007), IEEE, pp. 321–334.
2. Brandenburger, M., Cachin, C., and Knežević, N. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. *ACM Transactions on Privacy and Security (TOPS) 20*, 3 (2017), 8.
3. Buchegger, S., Schiöberg, D., Vu, L.-H., and Datta, A. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems* (2009), ACM, pp. 46–52.
4. Cachin, C., Keidar, I., and Shraer, A. Fork sequential consistency is blocking. *Information Processing Letters 109*, 7 (2009), 360–364.
5. Cachin, C., and Ohrimenko, O. Verifying the consistency of remote untrusted services with conflict-free operations. *Information and Computation 260* (2018), 72–88.
6. Civit, P., Gilbert, S., and Gramoli, V. Polygraph: Accountable byzantine agreement. *IACR Cryptology ePrint Archive* (2009), 587.
7. Crosby, S. A., and Wallach, D. S. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium* (2009), pp. 317–334.
8. Feldman, A. J., Blankstein, A., Freedman, M. J., and Felten, E. W. Privacy and integrity are possible in the untrusted cloud. *IEEE Data Eng. Bull. 35*, 4 (2012), 73–82.
9. Feldman, A. J., Blankstein, A., Freedman, M. J., and Felten, E. W. Social networking with frientegrity: Privacy and integrity with an untrusted provider. In *USENIX* (2012).
10. Feldman, A. J., Zeller, W. P., Freedman, M. J., and Felten, E. W. Sporc: Group collaboration using untrusted cloud resources. In *OSDI* (2010), vol. 10, pp. 337–350.
11. Goodrich, M. T., Lentini, J., Shin, M., Tamassia, R., and Cohen, R. Design and implementation of a distributed authenticated dictionary and its applications. Tech. rep., Technical report, Center for Geometric Computing, Brown University, 2002.

12. Goodrich, M. T., and Tamassia, R. Efficient authenticated dictionaries with skip lists and commutative hashing, Aug. 14 2007. US Patent 7,257,711.
13. Goodrich, M. T., Tamassia, R., and Hasić, J. An efficient dynamic and distributed cryptographic accumulator. In *International Conference on Information Security* (2002), Springer, pp. 372–388.
14. Graffi, K., Gross, C., Stingl, D., Hartung, D., Kovacevic, A., and Steinmetz, R. Lifesocial. kom: A secure and p2p-based solution for online social networks. In *2011 IEEE Consumer Communications and Networking Conference (CCNC)* (2011), IEEE, pp. 554–558.
15. Han, S., Shen, H., Kim, T., Krishnamurthy, A., Anderson, T., and Wetherall, D. Metasync: File synchronization across multiple untrusted storage services. In *2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15)* (2015), pp. 83–95.
16. Kurosawa, K., and Desmedt, Y. A new paradigm of hybrid encryption scheme. In *Annual International Cryptology Conference* (2004), Springer, pp. 426–442.
17. Li, J., and Maziéres, D. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI* (2007).
18. Loupasakis, A., Ntarmos, N., Triantafillou, P., and Makreshanski, D. exo: Decentralized autonomous scalable social networking. In *CIDR* (2011), pp. 85–95.
19. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (2011), 12.
20. Mazieres, D., and Shasha, D. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), ACM, pp. 108–117.
21. Narendula, R., Papaioannou, T. G., and Aberer, K. Privacy-aware and highly-available osn profiles. In *2010 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises* (2010), IEEE, pp. 211–216.
22. Nilizadeh, S., Jahid, S., Mittal, P., Borisov, N., and Kapadia, A. Cachet: a decentralized architecture for privacy preserving social networking with caching. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 337–348.
23. Palazzi, B. *Outsourced storage services: Authentication and security visualization.* PhD thesis, Ph. D. thesis, Roma Tre University, 2009.
24. Polivy, D. J., and Tamassia, R. Authenticating distributed data using web services and xml signatures. In *XML Security* (2002), pp. 80–89.
25. Shakimov, A., Lim, H., Cáceres, R., Cox, L. P., Li, K., Liu, D., and Varshavsky, A. Vis-a-vis: Privacy-preserving online social networking via virtual individual servers. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)* (2011), IEEE, pp. 1–10.
26. Strufe, T. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine 95* (2009).
27. Stuedi, P., Mohomed, I., Balakrishnan, M., Mao, Z. M., Ramasubramanian, V., Terry, D., and Wobber, T. Contrail: Enabling decentralized social networks on smartphones. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (2011), Springer, pp. 41–60.
28. Tamassia, R. Authenticated data structures. In *European symposium on algorithms* (2003), Springer, pp. 2–5.

29. Tegeler, F., Koll, D., and Fu, X. Gemstone: empowering decentralized social networking with high data availability. In *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011* (2011), IEEE, pp. 1–6.
30. Williams, P., Sion, R., and Shasha, D. E. The blind stone tablet: Outsourcing durability to untrusted parties. In *NDSS* (2009), Citeseer.