ACE: A Novel Software Platform to Ensure the Integrity of Long Term Archives[1]

(Technical Report UMIACS-TR-2007-07)

Sangchul Song and Joseph JaJa
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland, College Park

Abstract

We develop a new methodology to address the integrity of long term archives using rigorous cryptographic techniques. A prototype system called ACE (Auditing Control Environment) was designed and developed based on this methodology. ACE creates a small-size integrity token for each digital object and some cryptographic summary information based on all the objects handled within a dynamic time period. ACE continuously audits the contents of the various objects according to the policy set by the archive, and provides mechanisms for an independent third-party auditor to certify the integrity of any object. In fact, our approach will allow an independent auditor to verify the integrity of every version of an archived digital object as well as link the current version to the original form of the object when it was ingested into the archive. We show that ACE is very cost effective and scalable while making no assumptions about the archive architecture. We include in this paper some preliminary results on the validation and performance of ACE on a large image collection.

1. Introduction

One of the most challenging problems facing digital archives is how to ensure the authenticity of their holdings over the long term (tens or hundreds of years). Unless the authenticity of an archive can be assured, it would be almost impossible to use the archive's holdings to support any significant endeavor. Digital information is in general quite fragile, especially over time. Errors can be introduced because of hardware and media degradation, hardware and software malfunction, operational errors, security breaches, and malicious alterations, to name a few of the obvious ones. Other potential sources of errors, which are particularly relevant for long term archives, include major hardware and software systems changes due to technology evolution, and the possibility of major natural hazards and disasters such as fires, floods, and hurricanes. Two additional factors complicate this problem further. First, an object will typically be subjected to a number of transformations during its lifetime, including those migrative transformations due to format obsolescence. These transformations may alter the object in unintended ways. Second, most current integrity checking mechanisms are based on some type of cryptographic techniques, most of which are likely to become less immune to potential attacks over time and hence they will need to be replaced by stronger techniques. Therefore any approach to ensure the authenticity of a long term archive has to also be able to address these two problems as well.

Several technical approaches have been proposed to address the long term integrity of digital archives, including those that appeared in [2], [5], [8], [10], [12], [14], and [18], but none seems

[1] Supported in part by the National Science Foundation and the Library of Congress, contract IIS-0455995, under the DIGARCH program.

to offer a solid approach that is applicable to the different emerging architectures for digital archives (including centralized and distributed archives) and that is capable to continually monitor and verify the integrity of the data in a cost effective way.

In this paper, we introduce a general software environment called ACE (Auditing Control Environment), which is based on a rigorous cryptographic approach and yet quite efficient and can interoperate with any archiving architecture. Using the new framework, we introduce procedures to continually verify the integrity of the archive. Our approach will allow an independent auditor to verify the integrity of every version of an archived digital object as well as link the current version to the original form of the object when it was ingested into the archive.

Specifically, ACE is based on creating a small-size *integrity token* for each digital object upon its deposit into the archive (or upon registration of the object of an existing archive), to be stored either with the object itself or in a registry at the archive as authenticity metadata. Cryptographic summary information that depends on all the objects registered during a dynamic time period is stored and managed separately. The summary information is very compact and is of size independent of the number or sizes of the objects ingested. Regular audits will be continuously conducted, which will make use of the integrity tokens and the summary integrity information to ensure the integrity of both the objects and the integrity information. In our prototype, audits can also be triggered by an archive manager or by a user upon data access. However we are assuming that the auditing services are not allowed to change the content of the archive even if errors are detected. The responsibility for correcting errors is left to the archive administrator after being alerted by the auditing service.

In the next section, we provide an overview of integrity techniques, starting with bit stream integrity on disks and ending with the main methods that have been proposed for digital archives. In Section 3, a technical description of our approach is given and the basic verification procedures are introduced. The following section is devoted to a description of the overall architecture of our prototype including a description of the software components of our prototype. Section 5 presents workflows of our prototype. We conclude in Section 6 with a brief summary of the tests conducted on our prototype and some preliminary conclusions.

2. Existing Integrity Assurance Techniques

In this section, we describe some of the most common strategies used to ensure data integrity starting with the basic techniques for bit streams stored on various types of media or transmitted over a network.

Basic Techniques

Data residing on storage systems or being transmitted across a network can get corrupted due to media, hardware, or software failures. Disk errors, for example, are not uncommon and data on disk can get corrupted silently without being detected because a faulty disk controller causes misdirected writes [23]. This type of errors remains undetected because most storage software expects the media to function properly or fail explicitly rather than mis-operate at any point during its life time. The integrity of data can also get compromised because of software bugs. For example, data read from a storage device can get corrupted due to faulty device driver or a buggy file system which can cause data to become inaccessible [23]. Moreover, data integrity can be violated because of accidental use or operational errors. Unintended user's activity might cause the integrity to be broken. For instance, deletion of a file might lead to a malfunction of specific application/system software that depends on the accidentally deleted file. As a result of this action, integrity violations may occur.

The simplest technique for implementing integrity checks is to use some form of *replication* such as mirroring. The integrity verification can then be made by comparing the replicas against each other. This method can easily detect a change in the stored data only if the modification is not carried out in all the replicas and no errors are introduced during data movement. While this method is easy to implement and can be effective, it is quite expensive in terms storage use and time spent for comparisons, and has serious limitations in a distributed environment.

A well known approach used in RAID storage is based on coding techniques, the simplest of which is *parity checking* [21]. The parity across the RAID array is computed using the XOR logical function. The parity value is stored together with the data on the same disk array or on a different array dedicated to the parity itself. When the disk containing the data or the parity fails, the data or parity can *sometimes* be recovered using the remaining disk and performing the XOR operation [21]. The XOR parity is a very special type of *erasure codes*, which can be much more powerful. They all involve expanding the data using some types of algebraic operations in such a way that some errors may be detected and corrected. While these techniques are critical in maintaining some minimal level of data integrity on storage systems, they clearly fall far short to meeting our stated requirement for long term integrity of digital archives.

Another widely used method is based on *cryptographic hashing* (also called *checksum*) techniques. In this approach, a checksum of the bit-stream is computed and is stored persistently either with the data or separately. The checksum is calculated using a cryptographic hashing algorithm. In general, a cryptographic hash algorithm takes an input of arbitrary length and converts it into a single fixed-size value known as a *digest* or *hash value*. A critical property of cryptographic hash algorithms is that they are *one-way*, that is, given the hash value of a bit-stream A, it is computationally infeasible to find a different bit-stream B that has the same hash value [11], [19]. Assuming that the hash values are correct, data integrity can be verified by comparing the stored hash value with a newly computed hash from the data. The most common hash functions used in practice are MD5, SHA-1, SHA-256, and RIPEMD-160, none of which can be shown to be one-way functions but all of which seem to work well in practice (in spite of the recent attacks that illustrated how to break MD5 [25] and SHA-1 [26]). In addition to the one-way assumption, a key assumption of this technique is that the hash values can be stored securely with absolutely no changes introduced to these values over time. Such an assumption may be reasonable for maintaining integrity over brief periods of time but it is clearly not sufficient for digital archives especially as the number of objects (and hence the number of hashes) continues to significantly grow over time.

Techniques for Digital Archives

We now describe the most important methods that have been suggested for integrity verification for digital archives. These methods heavily depend on the basic architecture, organization, and policies assumed for the digital archive, and hence a straightforward comparison is not possible.

A straightforward method to address integrity checking for digital archives is to compute a hash for each object in the archive and store the hashes in a separate, secure and reliable registry (the hash could in addition be stored with the object as well). Integrity auditing involves periodic sampling of the content of the archive, computing the hash of each object, and comparing the computed hash with the stored hash value of the object. While such a scheme may be sufficient for small, centralized archives, it has some serious flaws for long term archives. Setting up a long term secure (centralized or distributed) registry that is expected to continuously grow, and maintaining its integrity over time, is highly non-trivial. In fact, this is somewhat the problem we are trying to solve, except that it is slightly simpler here as the hashes are smaller and less complex than the objects themselves. Another problem with this scheme is the fact that cryptographic hashing schemes are based on the one-way function assumptions that may not hold over time for the particular hashing scheme used, and hence they may need to be replaced with more powerful schemes. There is a way to address this particular issue (to be described later) but it is computationally expensive.

Another simple approach uses a combination of replication and hashing. In this approach, each digital object is replicated over a number of repositories. Integrity checking can be performed by computing the hash of each copy locally, and sending all the hashes to an auditor. A majority vote enables the auditor to discover the faulty copies, if any. This is the primary integrity scheme used in LOCKSS [18], which is peer-to-peer replication system for archiving electronic journals in which each participating library collects its own copy of the journals of interest. LOCKSS uses a peer-to-peer inter-cache protocol (LCAP) which is a cache auditing protocol. It runs LCAP continuously between all the caches to detect and correct any damage to cached contents. The process is similar to opinion polls in which all the caches vote. When a storage peer in LOCKSS calls for an audit of a digital object, each peer that owns a replica computes the corresponding hash value and sends back the value to the audit initiator. If the computed digest agrees with the majority of the replies, then the object is believed to be intact. Otherwise, the content has been tampered with, and the copy is discarded while a new copy is fetched from the publisher or one of the caches with the right copy. This approach is tied to the LOCKSS inherent peer to peer distributed architecture applied to a specific domain (with publishers having authentic copies of the objects), and hence does not address our more general framework. Moreover, note that LOCKSS nodes can arbitrarily initiate auditing requests thereby tying up distributed resources in an unpredictable way. In fact, a compromised LOCKSS node can initiate a denial of service attack that will affect all the other nodes. One can develop security techniques to counter such attacks such as in [2], but such techniques introduce an extra layer of complexity and additional costs. Note also that, in general, achieving consensus among distributed nodes that do not trust each other (and some of which may be faulty) is a difficult problem that has been studied extensively in the distributed computing literature. One can make use of Byzantine agreement strategies [14] but these are computational expensive and difficult to implement in a cost effective way in an environment such as LOCKSS.

Another possible approach is to make use of *digital signatures* based on *public key cryptography*. In essence, such a scheme involves a private-public key pair for performing signing/verification operations, and a supporting public-key infrastructure. The basic premise is that the private key is only known to the owner, and the public key is widely available. A message signed by a private key can be verified using the corresponding public key. The digital signature technology takes direct advantage of this property. The digital object is signed using the private key (note that the signature depends on the digital object *and* the private key), and anybody can verify the signature using the corresponding public key. If the verification process succeeds, the digital object is considered intact (and the identity of the author of the signature verified). Hence a possible approach to preserving the integrity of digital archives would be to sign each digital object using a private key only known to the archive. However the certificates (public keys signed by a widely trusted certificate authority) have a finite life with a fixed expiration date. Hence we need to have a trusted and reliable method to track the various public keys used over time. Also should the private key be compromised, the whole archive becomes at risk. In general, this is a difficult problem that can be solved using sophisticated techniques based on Byzantine agreement protocols and threshold cryptography [17], which shed serious doubts on its practicality in a production environment. Another potential problem with this scheme is its complete dependence on the secrecy of the private key of an independent party (certificate authority) – a risky proposition for long term archives.

We note that schemes based on public-key cryptography are computational intensive, especially when extended to complex, large objects. A common technique to reduce the cost is to compute a hash of the object, followed by a digital signature of the hash only. A somewhat related method was suggested in [2], which generates a Message Authentication Code (MAC) of each object, and then signs the MAC. The MAC can be generated through hashing of the object concatenated with a symmetric secret key. This scheme has the same weaknesses mentioned above about digital signatures (as well as other complications due to the maintenance of the secret key necessary to generate and validate the MAC).

We now introduce the time stamping technique, which plays a critical role in our proposed solution. A time stamp of a digital object D at time T is a record that can be used any time in the future (later than T) to verify that D existed at time T. The record typically contains a time indicator (date and time) and a guarantee (that depends on the time stamping service) that D existed in exactly this form at time T. It is clear that time stamping is essential for the long term integrity of digital archives since our integrity notion assumes an auditable record of all the versions of the object along the temporal domain beginning from the time the object was deposited into the archive up to the present. One way to implement time stamping is to through a Time Stamping Authority (TSA) that attaches a time designation to the object (or its hash) and signs it using the private key of the TSA. The British Library [12] uses this strategy through an independent TSA. The verification procedure depends completely on the trustworthiness of the TSA. We mentioned above a number of significant problems with any approach that uses digital signatures, which show up in this scheme as well. A second approach, and the one used in our solution, is based on *linked (or chained) hashing* [9], which amounts to cryptographically chaining objects together in a certain way such that a temporal ordering among the objects can be independently verified. In the next section, we will describe this technique as used in our approach and illustrate the automatic verification and auditing procedures that will ensure the long term integrity of digital information in a cost effective way.

3. Overview of the ACE Approach

We start by stating our underlying assumptions about the archive's environment, and then proceed to describe our technical approach.

Basic Assumptions

ACE does not make any assumptions about the architecture of the archive, which can be centralized, distributed, or peer-to-peer. However it seems reasonable to make the following assumptions regarding any long term archive.

Each object of an archive will be assigned a *globally unique identifier*, and the archive holds more than one copy of each object, one of which will be designated as the *master copy*. Each object retains its globally unique identifier even after it is subjected to a transformation. However the different versions of an object are distinguished by a certain versioning scheme (such as attaching a version number after each transformation), and all the versions are maintained (say in a dark archive) so that one can link the current version to all the previous versions down to the original version when the object was ingested into the archive.

An implicit implication of our assumptions is that a transformative migration or updating any of the metadata of an object (say as a result of repurposing, reorganization, or renewing the cryptographic information) will be performed on the master copy of the object, after which the archive can perform the versioning and replication operations according to its policies.

Technical Approach

The underpinnings of ACE consist of: (i) three types of integrity information; (ii) verification and auditing procedures; and (iii) methods to update integrity information as needed. We start by describing the three steps used to generate the integrity information, as follows.

The initial registration (typically during ingestion) of the objects is organized into rounds, each of which covers some time interval that is determined dynamically. A typical time stamping service (several of which are already available on the internet) operates using fixed intervals with the granularity of one second for each round. In our prototype, the time interval is dynamically adjusted, depending on the number of registration requests, so as to ensure fixed small-size integrity information. Typically, our time interval ranges from one second to at most an hour. During each round, the hashes of all the objects submitted for registration as well as random hashes as necessary are aggregated using an *authentication tree* such as Merkle's tree [20]. The value at the root is a hash value that depends in a cryptographic sense on all the objects processed during a round. For each object, we assemble a short list of hashes from the tree, called a *proof*, which constitutes the first type of cryptographic information, to be called an *integrity token*, stored with the object (as authentication metadata) or in a separate registry at the archive (for distributed archives, the integrity token will appear at any node where a replica of the object is stored).

The second step consists of linking the hash value generated at each round with the hash values generated at the previous rounds using a structure that depends on the linking scheme used. In our prototype, we use a simple binary linking scheme that computes the hash value of the previous *Cryptographic Summary Information* (CSI) concatenated with the hash value of the current round. This is the same scheme as suggested in [9] and [24].

The third step consists of aggregating the global hash values over each week using an authentication tree. The value generated at the root of the tree is called a *witness* for that particular week, and is published on the web as a widely observed witness. We explain later how this is accomplished on our current prototype. This value is also stored on a CD-ROM (in fact, on multiple CD-ROMs that are refreshed on a regular basis).

Before providing more details about each of the steps, we note that the integrity token pertains to a unique object; the CSI pertains to a unique round but depends on all the objects submitted at that round and previously computed round values; and the witness pertains to a unique week but depends in a cryptographic sense on all the CSI values computed at all the rounds during that week. Moreover the size of each type is small and fixed and depends on the hash functions used – typically in the order of a few kilobytes at most. Clearly the number of integrity tokens is equal to the number of objects while the numbers of CSI values and witness do not depend on the number of objects ingested. In our prototype, there are 52 witness values per year.

We now provide more details about each of the steps. During the first step, the hash values of all the objects processed during a round form the leaves of a balanced binary tree such that the value stored at each internal node is the hash value of the concatenated hashes at the children. Note that we typically insert random hash values into each round, which also ensures that there always will be a certain minimal number of objects in each round. The value computed at the root of the tree is the hash value of the round. Each object will retain a proof of its participation in this round, which consists of the hash values of the siblings of all the nodes on the unique path to the root, plus the previous CSI. Consider for example a round involving eight objects with the hash values $h_0, h_1, h_2, ..., h_7$. The corresponding authentication tree is shown in Figure 1.

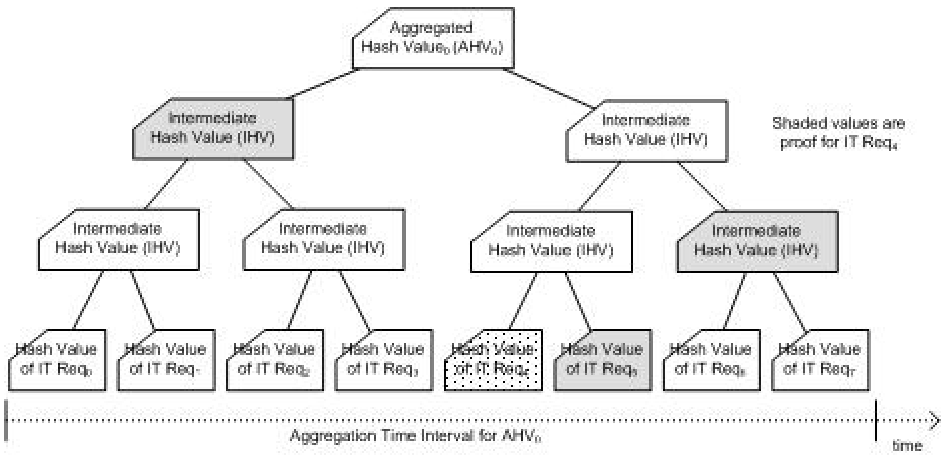

Figure 1. Authentication Tree (IT Req$_i$ contains h$_i$)

The CSI values are chained in a simple binary chain as illustrated in Figure 2. Therefore the CSI generated at time interval *t* cryptographically depends on the hashes of all the objects ingested into the system at any time less than or equal to *t*.

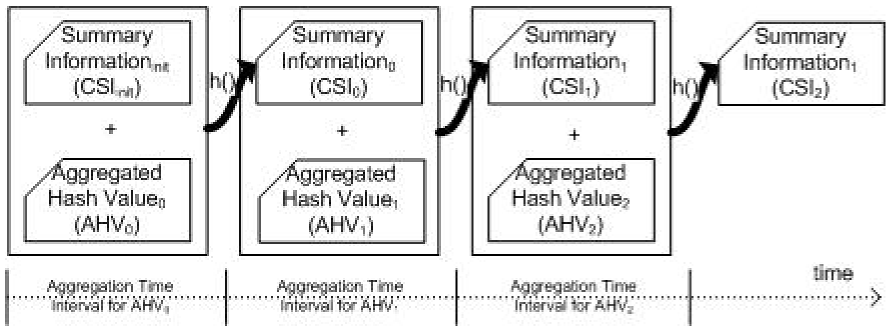
Summary Information_init (CSI_init)
+
Aggregated Hash Value₀ (AHV₀)
h()
Summary Information₀ (CSI₀)
Aggregated Hash Value₁ (AHV₁)
Summary Information₁ (CSI₁)
Aggregated Hash Value₂ (AHV₂)
Summary Information₁ (CSI₂)
Aggregation Time Interval for AHV₀
Aggregation Time Interval for AHV₁
Aggregation Time Interval for AHV₂
time

Figure 2. CSI Chain

The witness value corresponding to each week is generated using Merkle tree built on the CSI values generated during this week and randomly generated hashes.

Verification and Auditing Procedures

ACE provides two types of integrity auditing, the first involves a process running on a moderately secure server external to the archive, which verifies the integrity of the archive's content in a periodic, regular fashion; the second involves an auditing process triggered by an archivist or by a user upon data access. Details of how these processes are implemented in our prototype system will be given in Section 4. Here we mention the verification algorithms used to perform the auditing. The correctness of our algorithms depends on two assumptions: the hash functions have the collision intractability property, and the witness values cannot be compromised. Should one of the hash functions be compromised, the integrity information needs to be updated in such a way that we can verify the authenticity of the data from the initial ingestion up to the present. We will describe in the next subsection how to accomplish this. As for the witness values, their total size over a period of ten years is less than 1-2MB, and hence a web publishing mechanism backed-up with storage on CD-ROMs (and coupled with majority voting whenever necessary) should be sufficient to ensure its integrity. We will explain in Section 5 the web publishing mechanism used in our current prototype.

As stated above, we generate three types of integrity information – a token attached to an object, a CSI corresponding to a round, and witness value W corresponding to a weekly digest of all the CSI values generated during that week. A simple way to verify the integrity of an object involves the following steps.

Step 1. We start by ensuring the correctness of the object's integrity token. We use the hash value in combination with the proof in the integrity token to determine the CSI of the round during which the object was registered into the system. If the computed CSI is the same as the CSI stored at the registration time, it indicates that the token is intact, and we proceed to Step 2. Otherwise, we stop the verification procedure, and notify the archive manager of the possible corruption of the integrity token. In this case, the archive manager may verify the integrity of the object by comparing it to a replica, and re-register the object to create a new integrity token.

Step 2. Once we are confident that the integrity token is intact, we verify the integrity of the object as follows. We compute the hash of the given object and compare it to the hash stored in the object's integrity token. If there is an agreement, we conclude that the object is intact. Otherwise, the object is corrupt.

We now outline the process to verify the integrity of the CSI values. For each witness value W and for each CSI value computed within that week, we check to see if the proof attached to the CSI value yields W. In the affirmative, the CSI value is correct. Otherwise, it is not.

We now address the issue of what to do in the case when the object or the CSI is determined to be incorrect. In our prototype, our verification service notifies the archive manager about the faulty object, and it is left up to the archive to take the appropriate action. We believe that an integrity verification service should not be allowed to modify anything in the archive. Its main function is to continually monitor and verify the authenticity of the data. In our experimental setting, we use the distributed persistent archive pilot system that provides at least three replicas for each digital object based on the federated SRB (Storage Resource Broker) grid technology [3], and hence a copy can be corrected using a voting scheme over the distributed archive. As for correcting erroneous CSI values, our integrity checking prototype makes use of a three-way mirrored registry of the CSI values, each of which is audited independently. Hence the faulty CSI can be corrected using a correct replica from the other registries. Note that the size of a registry grows in the order of a few gigabytes per year (independent of the size of the archive), and that the registry is not publicly accessible. Therefore maintaining the integrity of the CSI registry can be done in a cost effective way.

Updating Integrity Information

There are two cases in which the integrity information must be updated. The first case is when the archive decides to substitute a stronger hash function for one of the hash functions currently in use because of some recently discovered potential threats. The second is when the archive decides to apply certain transformations to some of the objects (because of the possibility of a format becoming outdated for example). There is a well-known solution to deal with renewing the integrity information for the first case by re-registering each related object with the old integrity token attached to it (see for example [8]). Such a solution will ensure our ability to verify the integrity of the object since its ingestion into the archive as articulated in earlier work. This process increases the size of the integrity token, but has no impact on the sizes of the other integrity components.

We now discuss how to renew the integrity information in the case when the object is subjected to a transformation. A possible solution would be to re-register the new object by concatenating the hashes of the old and the new form of the object and an ID of the transformation, and use the resulting string as if it were the hash of an object to be registered. However, as mentioned before, we are assuming that all the versions of each object are maintained by the archive (using a deep archive for example) and each version will include the information about each transformation. Therefore it would be sufficient in this case to include the version number in addition to the hash of the object and re-register the object after the transformation to generate new integrity information for this particular version of the object. Different versions can be linked through the global ID of the object using the dark archive, and hence it is possible to verify the integrity of all the versions of each object starting with the current one and ending with the first version ingested into the archive. Note that the integrity of an object should be verified before it is transformed into a new format to ensure its authenticity at this time of its history.

4. Prototype System Architecture

The ACE prototype includes two major components: ACE Integrity Management System (ACE-IMS) and ACE Audit Manager (ACE-AM). The ACE-IMS is a server that issues integrity tokens, preserves the CSI values, and computes and publishes the witness values. The ACE-AM is a bridging component between the archive and the ACE-IMS, which is local to each archiving node. In a distributed setting, the audit managers work asynchronously independent of each other, and hence copies of the same object will be audited independently of each other.

The ACE-IMS, operating separately from the archive, provides two important services: integrity token issuing and CSI verification. The former service generates an integrity token upon a request from the archive. Using the digital object and the integrity token, the archive can at anytime construct the cryptographic summary corresponding to the round in which the digital object was registered. The CSI values will be maintained separately and independently by the ACE-IMS.

In a typical archiving environment, the integrity tokens can be stored either with the object itself or in a separate registry dedicated to authenticity metadata. In our prototype, we use a separate database to hold the integrity tokens.

The ACE Audit Manager (ACE-AM) is local to an archiving node whose main function is to pass information between the archiving node and the ACE-IMS. In particular, the ACE-AM selects a digital object to be audited, either based on the local periodic auditing policy of the archiving node or upon request from an archive manager or a user. It then retrieves the digital object’s integrity token, computes the hash of the object, and sends this information to the ACE-IMS.

Figure 3 shows the overall ACE architecture assuming a distributed archiving infrastructure. A centralized archiving infrastructure will reduce to a single archiving node. The upper section represents the archive, the middle section contains the ACE-AM that is local to each archiving node, and the lower section represents the ACE-IMS, which is completely outside the archive. The implementation details of the ACM-IMS and the ACM-AM will be given next.

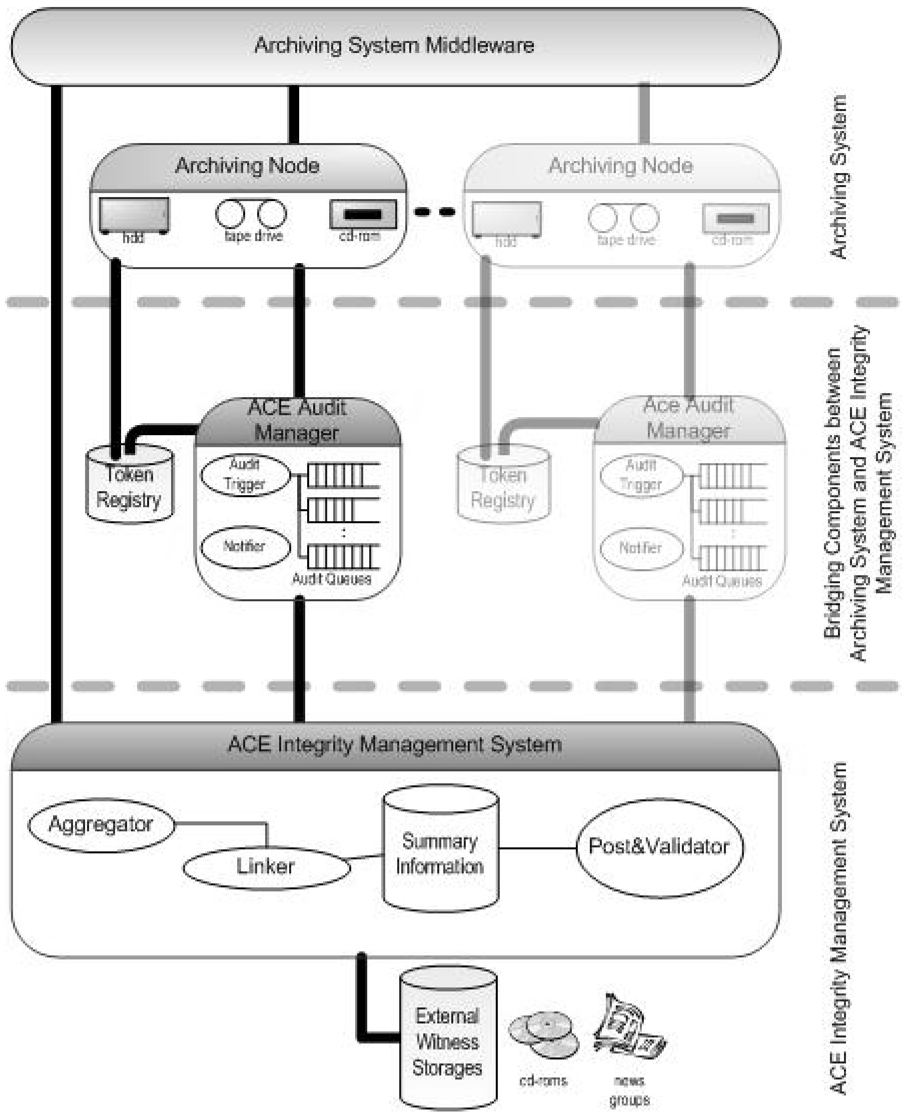

Figure 3. ACE System Architecture

Software Components

We now provide the details of the ACE-IMS and the ACE-AM, including a description of their internal modules. We also cover the communication mechanisms between the ACE-IMS and its users. Our implementation largely relies on open standards and web technologies. In particular, we have chosen Java as the base programming language and SOAP [29] as the main communication protocol. We use XML to represent various integrity information.

ACE Integrity Management System (ACE-IMS)

The ACE-IMS generates the integrity tokens, the CSI for each round, and the weekly witness values, and provides services to the archive. The WSDL specifications of these services are given in Appendix A. We start by describing three main services that are provided by the ACE-IMS (Figure 4 shows the message formats used) followed by a description of internal modules of this component.

- TSS_Stamp(*TS_Req*) : This function, called upon to register an object to ACE, adds this request to the aggregation queue, which will be processed by the Aggregator module (to be explained soon) to generate the integrity token for the request. The function returns the registration receipt (TS_Rcpt) that, among others, contains the request ID and the expected time when the integrity token will be ready.

- TSS_ITRequest*(RequestID)* : This function returns the integrity token that corresponds to the request ID (*RequestID*), if available. The function is typically called by the same entity that previously called TSS_Stamp when the expected time is reached.

- TSS_CompareCSI(*TimeStamp*, *Csi*): This function compares the supplied cryptographic summary information, *Csi*, to the one maintained by the ACE-IMS, and indexed by *TimeStamp*. The function returns true or false depending on whether or not the two agree. The auditor computes its own value Csi of CSI from the current digital object and its integrity token, and then calls this function to verify the integrity of the object.

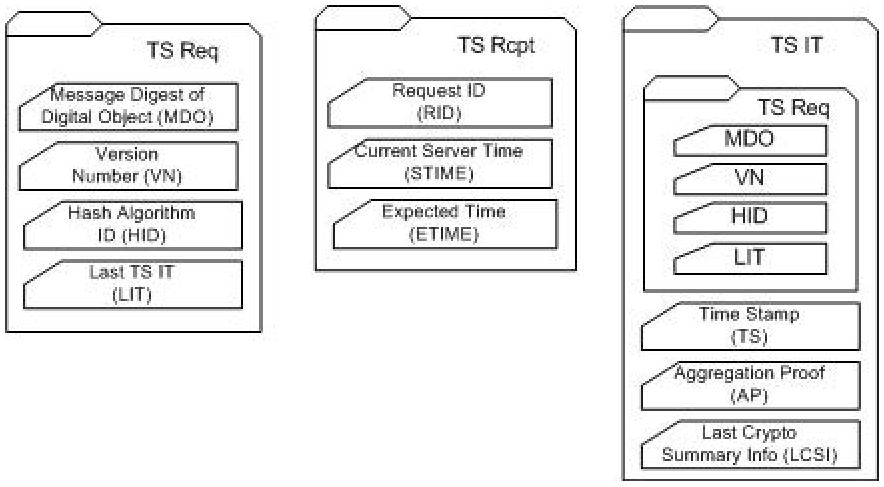

Figure 4. Message Formats

We now give details about each of the internal modules.

- **Aggregator**: The Aggregator is responsible for managing the rounds of aggregation and building an authentication tree (Figure 1) given the registration requests. The time interval of each round is determined by two internal elements: a timer and a traffic monitor. If there is at least one unanswered registration request in the aggregation queue, the timer sends a trigger signal at a fixed interval (currently every hour). On the other hand, the traffic monitor sends a trigger signal when the number of unanswered registration requests exceeds a certain number (say 1024). Regardless of the sender, whenever a trigger signal is received, the current round of aggregation is closed and the next round of aggregation begins. By using these two conditions (the amount of time and the number of requests), not only can the Aggregator control the size of each integrity token (or more specifically the length of a proof in an integrity token), it can also ensure that every request will be answered in a timely manner (currently within an hour at most). The dynamic time window is expected to be particularly beneficial in an archiving environment that is likely to exhibit bursty behavior. The time when an aggregation round ends is the timestamp value assigned to all the registration requests during the round.

- **Linker**: The Linker is responsible for generating and storing the CSI for each aggregation round. The root of the authentication tree that was built after a round of aggregation is concatenated to the previous CSI, which is then hashed (using possibly a different hash function than the one used for aggregation) to generate a new CSI for this aggregation round (Figure 2). The new CSI is stored along with the timestamp of the round in a database. An integrity token containing the time stamp of the round, the proof to the authentication tree, and the previous CSI is built. The timestamp is used by an auditor to query the ACE_IMS about the CSI of the round associated with the timestamp, whereas the proof and previous CSI are used by the auditor to calculate its own value of CSI. When the two CSI values match, the auditor concludes that the digital object has not been altered.

- **Post & Validat**e: The Post & Validate module creates, posts, and validates witness values. To create a witness, we use the same method used in Aggregator, that is, the CSI values produced during a week period are aggregated together in an authentication tree. The root of the authentication tree is the witness value, which is published over the internet through public, widely known entities. ACE currently uses the Internet newsgroups at Google, Yahoo, and MSN to publish the witness values. We are also planning to use public digital library services to publish these values. Since the size of the witness is very small (1KB~2KB a year when SHA256 is used to build the authentication tree), we also store them on a CD-ROM.

ACE Audit Manager (ACE-AM)

The ACE-AM is a client of the ACE-IMS which initiates both regular and on-demand audits. The regular audits are automatically performed according to a policy set by an administrator at the corresponding local archiving node, whereas the on-demand audits are triggered by an archive manager or by a user upon data access. In the following, we discuss the ACE-AM's two internal modules: Audit Trigger and Notifier.

- **Audit Trigger:** As mentioned before, two types of audits are executed by the Audit Trigger: regular audits and on-demand audits. Upon receiving an on-demand audit request, the Audit Trigger immediately activates the audit on the given object. On the contrary, regular audits are continuously and autonomously conducted. For regular audits, the Audit Trigger manages a number of audit queues. Each audit queue can be scheduled to be audited with a certain rule. For example, the group of objects stored in an old hard drive can be assigned to a higher priority queue than objects stored in more reliable storage. For each queue, the Audit Trigger performs the following steps for each object in the queue at the appropriate time.

  Step 1. The Audit Trigger retrieves the corresponding integrity token either directly from the object, or from a separate local registry.

  Step 2. Using the proof in the integrity token, the Audit Trigger computes the CSI and compares it to the stored CSI in the ACE-IMS by calling the web service function TSS_CompareCSI(*TimeStamp*, *Csi*). A match indicates that the integrity token is intact and the Audit Trigger proceeds to Step 3. Otherwise, the Audit Trigger informs the Notifier of the possible corruption of the integrity token.

  Step 3. If the integrity token is verified to be intact in Step 2, the Audit Trigger computes a hash of the given object and compares it to the one in the integrity token. If they match, the object is intact. Otherwise the object is corrupt and the Audit Trigger reports the Notifier about the corruption of the object.

  Upon finishing the above steps, the Audit Manager continues by pulling the next object out of the audit queue at the next appropriate time unit.

Note that the decision about the integrity of the given object is made only in Step 3, and note also that the object is determined to be intact only after passing the hash comparison test in Step 3. In all the other cases, an appropriate message is constructed and passed to the Notifier.

Once informed by the Audit Trigger, the Notifier performs the following actions.

- ***Notifier:*** The Notifier warns the archive administrator, or its automated counterpart, of either a corrupted object or a (possibly) corrupted integrity token. Since we are assuming that the auditing services are not allowed to change the content of the archive, the ACE-AM does not attempt to make any corrections. Instead, once alerted by the Audit Trigger, the Notifier builds an alert message that contains the Audit Trigger's report, and delivers the message to the archive administrator who can then initiate a comparison of the object with other copies in the archive or trigger an audit of the ACE-IMS using witnesses or follow other procedures depending on the policy set by the archive to handle such problems.

5. Workflows

In this section, we give a description of the ACE workflows executed during object registration, witness publication, auditing, and witness validation.

Object Registration

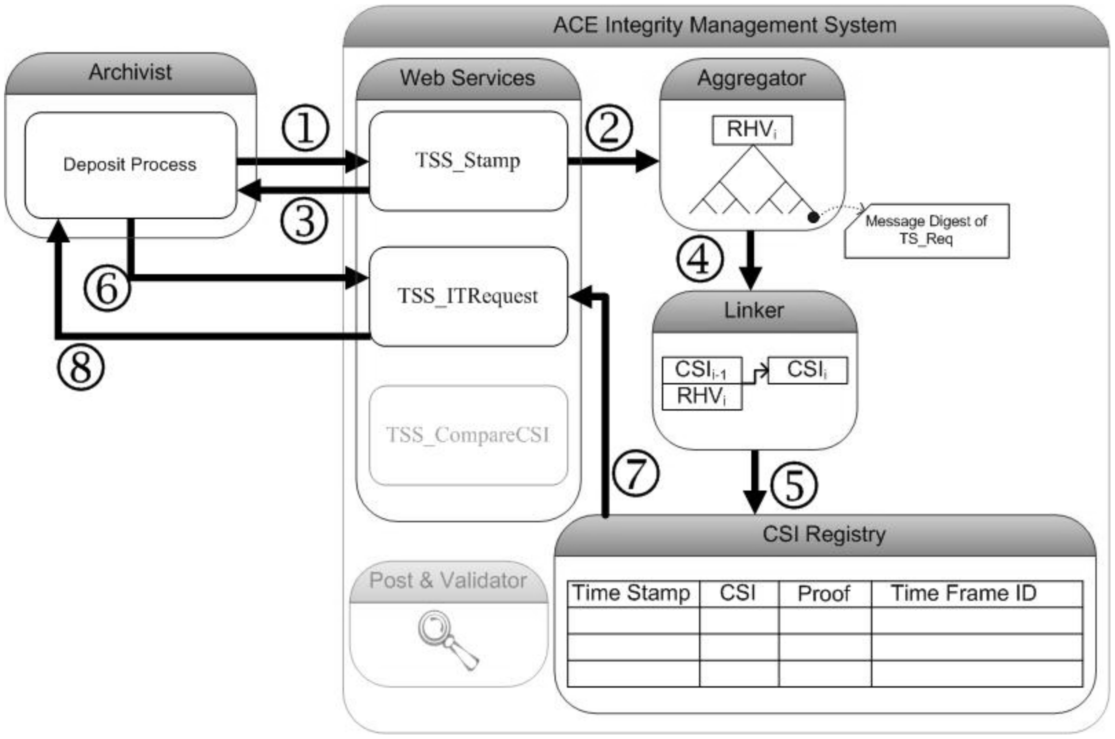

Figure 5 illustrates the registration workflow initiated during the last phase of the ingestion process of an object (or could be applied to an object that is already in the archive).

Step 1. During the last phase of the ingestion process, the deposit process (DP) initiates a request for an integrity token of the digital object by calling TSS_Stamp service of ACE-IMS while providing the message digest of the digital object as input.

Step 2. The ACE-IMS queues the request into its aggregation queue to be later processed by the Aggregator.

Step 3. The ACE-IMS sends back a receipt that contains a request id (REQID) corresponding to the request, and the expected time (ETIME) when the integrity token will be ready.

Step 4. Once the round time is up, the Aggregator gathers all the requests received, inserts random hashes if necessary, and builds an authentication tree.

Step 5. Using the root value of the aggregation tree and the previous CSI, the Aggregator computes the CSI for the current aggregation round, and stores the CSI into the CSI Registry along with the current time. Using this time value as the Time Stamp, the current CSI, and the proof of the authentication tree, the integrity token for each request is individually built.

Step 6. When ETIME has reached, DP calls TSS_ITRequest with the REQID asking for the integrity token.

Step 7. The integrity token made in Step 5 is retrieved.

Step 8. The integrity token for the digital object is finally sent back to the requester (DP).

Once the integrity token is received, it is attached to the object after which the object is deposited into the archive following the archive's policy (which may include creating a number of replicas, and storing the complete object into a deep archive).

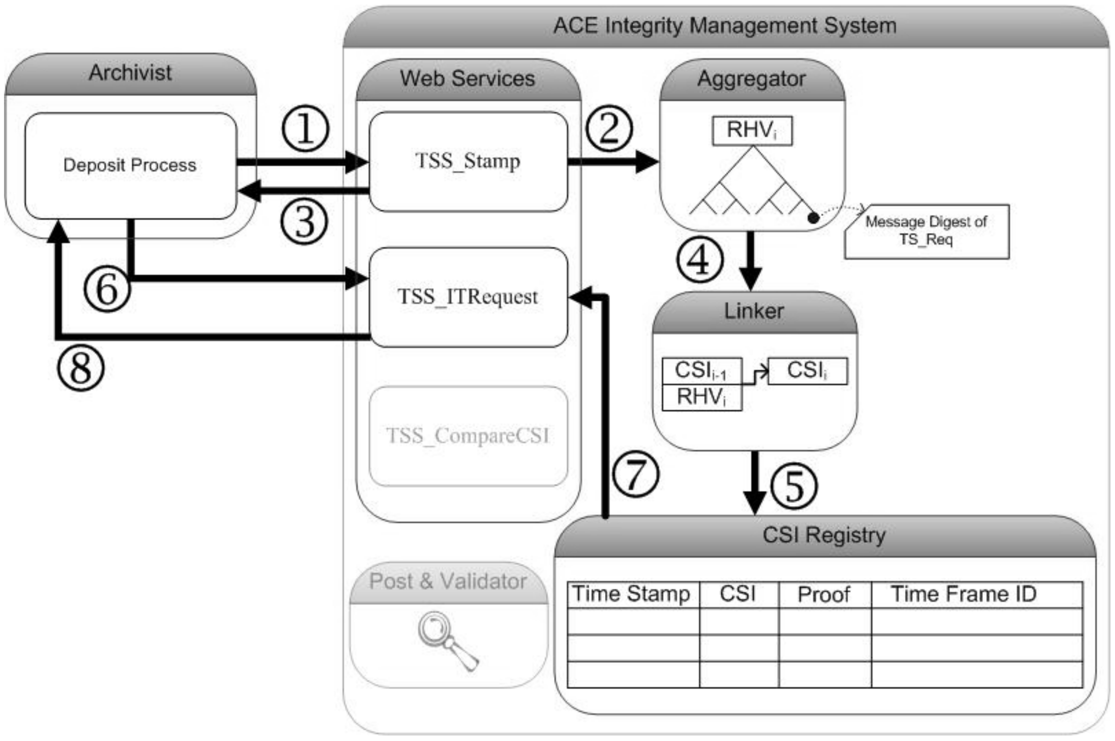

Figure 5. Object Registration Workflow

Witness Publication

The ACE-IMS periodically (currently once a week) runs another aggregation round for computing a witness for that particular period. The purpose of the witness is to ensure the integrity of CSI values generated during the various rounds during a week. Figure 6 shows the corresponding workflow.

Step 1. Once a week, the Post & Validate module builds an authentication tree from the CSI values generated during that week, using the same technique as before. The root value of the authentication tree concatenated with the witness from the previous week is fed into a hash function (currently SHA-256) to create the witness of the week. The proof of the authentication tree and the current time is written back to the CSI registry in the Proof field and the Time Frame ID, respectively.

Step 2. The witness of the week currently gets posted to the newsgroups at Google, Yahoo and MSN. These newsgroup services will also automatically send emails to all the newsgroup subscribers with the witness value. In an operational setting, we expect these witness values to be published through widely known library or storage services. The witness is also saved on a CD-ROM.

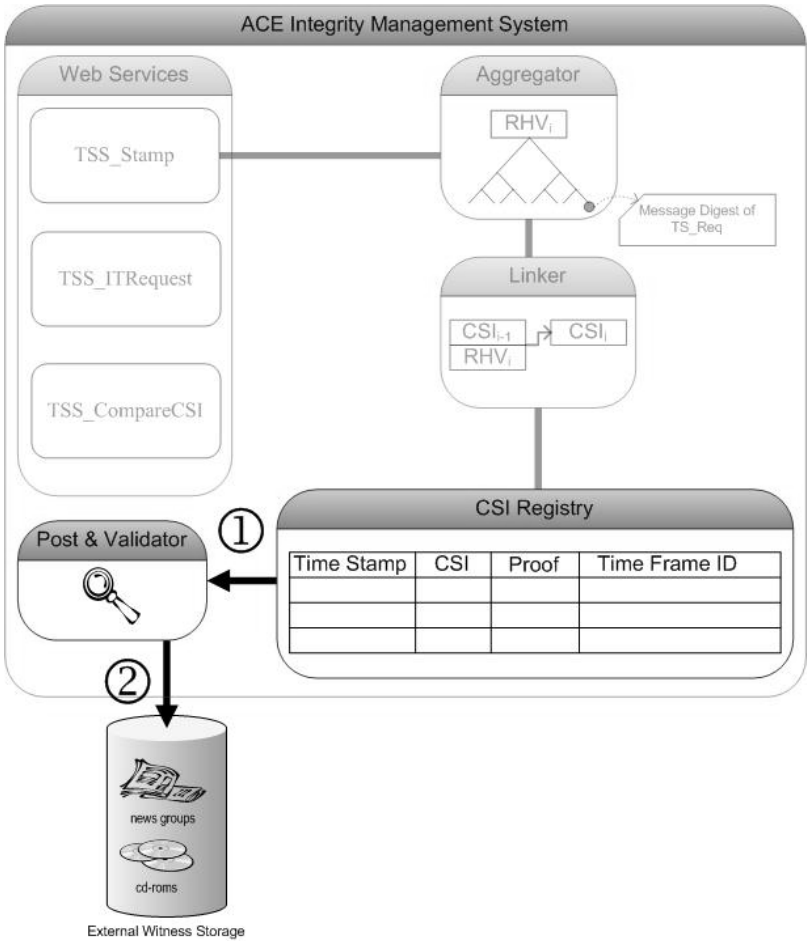

Figure 6. Witness Publication Workflow

Audit

The ACE-AM (ACE Audit Manager) is responsible both for regular and on-demand audits. The digital object to be audited is determined by the Audit Trigger scheduler for regular audits, or by the archive manager or the user for on-demand audits. Figure 7 illustrates the workflow of the process to audit an object.

Step 1. The ACE-AM retrieves the integrity token associated with the digital object and computes the corresponding CSI value from the integrity token, followed by calling TSS_CompareCsi using the computed value and the timestamp parameters.

Step 2. The ACE-IMS retrieves the stored CSI that corresponds to the timestamp, and compares the stored CSI to the computed value.

Step 3. The comparison result is returned to the ACE-AM.

Step 4. If the comparison result is positive, the Audit Manger computes the hash of the object and compares it to the hash stored in the integrity token.

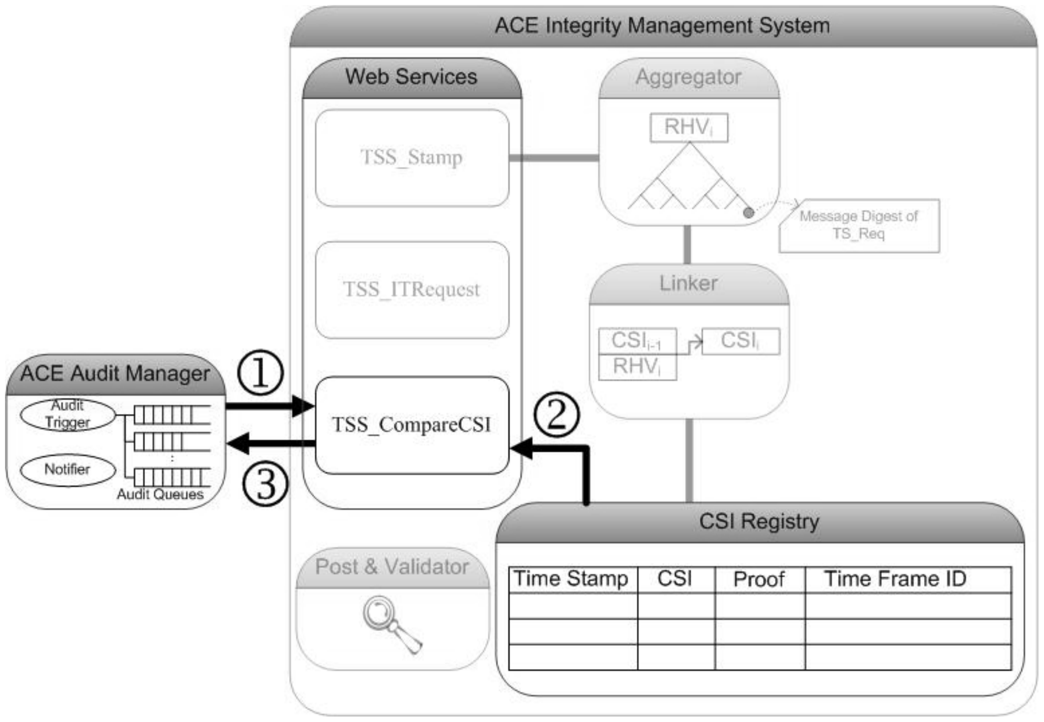

Figure 7. Audit Workflow

Depending on the comparison results performed in Step 2 and Step 4, the ACE-AM either successfully finishes the audit on the digital object, or lets the Notifier deliver an alert message to the archive manager.

If an alert message is delivered to the archive administrator, the administrator can take the appropriate steps to deal with the suspicious object as per the set policies of the archive.

Witness Validation

The Post & Validate module runs periodic internal audits on the CSI values. The validation is based on the public witness values, each of which cryptographically represents a set of CSI values. For each set of CSI values within the same Time Frame, Figure 8 shows the witness validation workflow.

Step 1. From the CSI Registry, the Post & Validate gathers the CSI values that share the same Time Frame ID, and builds an authentication tree using the same technique that produced the original witness. The root of the authentication tree is concatenated with the previous witness, which is saved from the previous witness validation process. This concatenated value is hashed to generate a validation witness.

Step 2. The Post & Validate retrieves the published witness of this Time Frame ID from the Internet. The published witness is then compared to the validation witness. The validation fails if the two witnesses are different, in which case all the CSI values within this Time Frame ID are marked invalid.

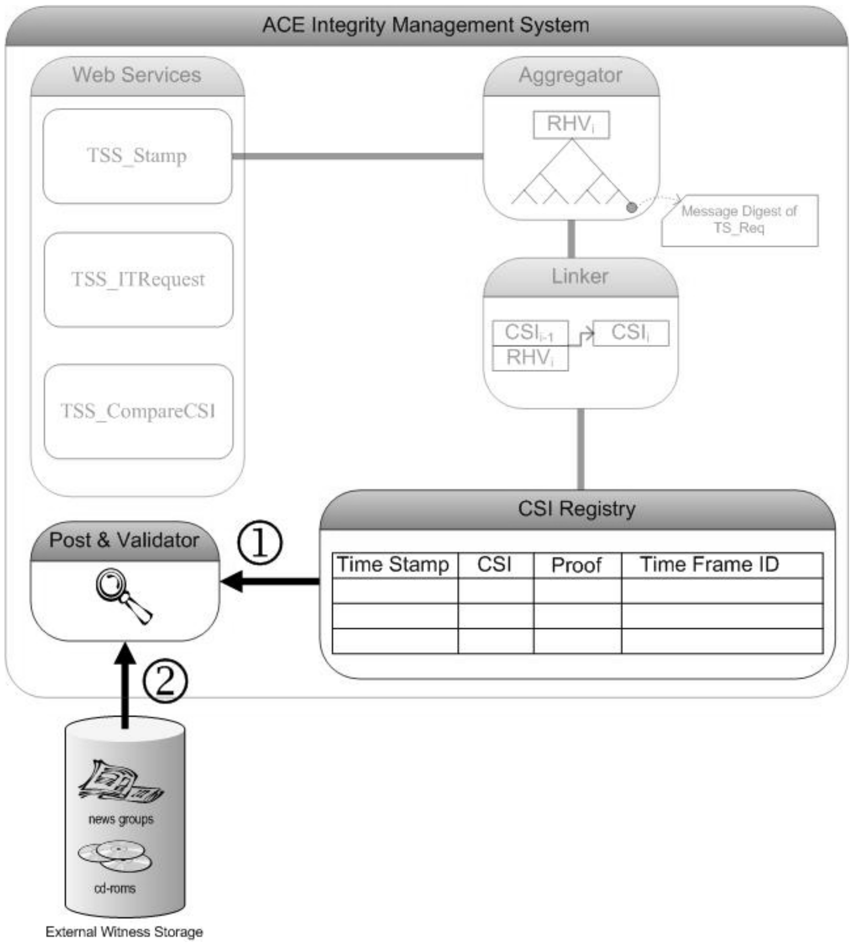

Figure 8. Witness Validation Workflow

Once CSI is marked invalid, the audit on the digital objects that belong to the CSI will fail (Step 1 in Section 4 – ACE-AM – Audit Trigger), in which case ACE will forward its alert message to the archive administrator who will then take the appropriate steps. For example, the archive administrators can simply re-register the digital objects to ACE.

6. Conclusion

We have designed and implemented a digital archive integrity management system that is based on time stamping and linked hashing techniques. This system is third-party auditable, cryptographically rigorous yet cost-effective, update-aware, highly interoperable, and scalable.

Specifically, ACE creates and maintains a small size integrity token for each digital object, which can later be used to rigorously audit the digital object. Combined with other cryptographically related data (CSI values and witnesses), an integrity token can provide convincing evidence whether or not the associated digital object is intact, not only to our own auditor (ACE-AM), but also to any independent third-party auditor.

Although our approach relies on rigorous cryptographic techniques, ACE is quite cost-effective since it does not demand an expensive external infrastructure, such as a PKI infrastructure, which significantly reduces the operational and computational overhead.

ACE can cope with the future transformations of the digital object or changes in the cryptographic functions. During the course of the update, The ACE’s update scheme includes a link to the previous data (the previous digital object, the previous integrity token and/or the previous hash function) in such a way that a future audit can certify not only the integrity of the current digital object but also the integrity of every version of the digital object since its ingestion into the archive.

ACE can interoperate with any archiving architecture since it does not rely on any assumption about the archive architecture. In addition, ACE is platform independent and is built using exclusively open standards and web technologies.

ACE is efficient and scalable as well. The ACE Audit Managers can be distributed among the archiving nodes if these are distributed, each performing its tasks independently of the others. While the web-based ACE Integrity Management System is centralized, its function is relatively quite simple and can be performed extremely quickly, and is only available to the participating archive. Simple well-known web server techniques can be used to ensure extremely high availability, and scalability ([1], [14], and [5]). Moreover, the aggregation scheme with adaptive round time is expected to help reduce the amount of work that the ACE-IMS needs to deal with at a given time.

In terms of performance, we have evaluated ACE using the NARA EAP (The National Archives Electronic Access Project) Image Collection consisting of approximately 130,000 files of total size over 1.1TB. We were able to fully audit all the objects in about 15 hours while storing the data remotely on a separate server. Most of the time was spent in moving the data between the separate machines in our environment. We expect performance to be much better in a production environment since all the data movement will be carried out locally between the audit manager and the local storage.

References

[1] D. Anderson, T. Yang, V. Holmedahl, and O.H. Ibarra. "*SWEB: Towards a Scalable World Wide Server on Multicomputers*". In Proceedings of IPPS'96, April 1996

[2] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In Bruce Christianson, Bruno Crispo, and Mike Roe, editors, Proceedings of the 8th International Workshop on Security Protocols, to appear in the Lecture Notes in Computer Science series, Cambridge, UK, April 2000. Springe

[3] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In Procs. of CASCON'98, Toronto, Canada, 1998.

[4] J. Buchmann, A. May, and U. Vollmer, “Perspectives for Cryptographic Long-Term Security,” Communications of the ACM, 49(9), September 2006, 50-55.

[5] O. P. Damani, P. Y. Chung, Y. Huang, C. Kintala, and Y. M. Wang, “ONE-IP: Techniques for hosting a service on a cluster of machines,” in *Proc. the Sixth Int. World Wide Web Conference*, April 1997.

[6] Alan O. Freier, Philip Karlton, and Paul C. Kocher. “The SSL Protocol – Version 3.0,” Internet Draft, Transport Layer Security Working Group, November 1996.

[7] H.M. Gladney. “Trustworthy 100-Year Digital Objects.” ACM TOIS,22(3), July 2004

[8] Stuart Haber and Pandurang Kamat. “Content Integrity Service for Long-Term Digital Archives.” In Proceedings of Archiving 2006, May 2006, pp 159-164.

[9] Stuart Haber and W. Scott Stornella, “How to time-stamp a digital document,” Journal of Cryptology, 1991.

[10] Ronald Jantz and Michael J. Giarlo. “Digital Preservation – Architecture and Technology for Trusted Digital Repositories Reich.” *D-Lib Magazine*, 7(6), June 2005. <http://www.dlib.org/dlib/june05/jantz/06jantz.html>

[11] Charlie Kaufman, Radio Perlman, and Mike Speciner, Network Security – Private Communication in a Public World, Prentice Hall, Second Edition, 2003.

[12] Lisa Kelly. “British Library secures integrity of digital archive.” Computing. 25 Apr 25 2006. <http://www.computing.co.uk/computing/news/2154704/british-li>

[13] John Kubiatowicz, et. al. “OceanStore: An Architecture for Global-Scale Persistent Storage.” In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000, pp 190-201.

[14] T. T. Kwan, R. E. McGrath, and D. A. Reed, “NCSA's World Wide Web Server: Design and Performance”, *IEEE Computer*, pp. 68-74, Nov. 1995

[15] Leslie Lamport , Robert Shostak , Marshall Pease, The Byzantine Generals Problem, ACM Transactions on Programming Languages and Systems (TOPLAS), v.4 n.3, p.382-401, July 1982

[16] Clifford A. Lynch. "Authenticity and Integrity in the Digital Environment: An Exploratory Analysis of the Central Role of Trust." Authenticity in a Digital Environment (Washington, DC: Council on Library and Information Resources, 2000), pp 32-50. <http://www.clir.org/pubs/reports/pub92/lynch.html>

[17] Petros Maniatis, T.J. Giuli, and Mary Baker, "Enabling the Long-Term Archival of Signed Documents through Time Stamping," May 2006.

[18] Petros Maniatis, TJ Guili, David S. H. Rosenthal and Mary Baker. "THE LOCKSS Peer-to-peer Digital Preservation System." ACM TOCS, 23(1), February 2005.

[19] Alfred Menezes, Paul van Oorschot, and Scott Vanstone. Handbook of Applied Cryptography, CRC Press, 1996.

[20] Ralph Merkle. "Protocols for public key cryptosystems." In Proceedings of the 1980 Symposium on Security and Privacy, IEEE Computer Society Press, 1980, pp 122–133.

[21] D. Patterson, G. Gibson, and R. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." In the proceeding of SIGMOD, PP. 109-116, June 1988

[22] James S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems." Software -- Practice & Experience, 27(9), September 1997, pp. 995-1012.

[23] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok." Ensuring Data Integrity in Storage: Techniques and Applications." ACM TOS, 2005

[24] Surety, Inc. <http://www.surety.com/>

[25] Xiaoyun Wang and Hongbo Yu. "How to break MD5 and other hash functions." In Ronald Cramer, editor, Advances in Cryptology – EUROCRYPT 2005, Volume 3494 of Lecture Notes in Computer Science, 2005.

[26] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding collisions in the full SHA-1." In Victor Shoup, editor, Advances in Cryptology — CRYPTO 2005, volume 3621 of Lecture Notes in Computer Science, 2005.

[27] Hakim Weatherspoon and John Kubiatowicz. "Erasure Coding vs. Replication: A Quantitative Comparison." In Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), March 2002, pp 328-338.

[28] Hakim Weatherspoon and John Kubiatowicz. "Naming and Integrity: Self-Verifying Data in Peer-to-Peer Systems." In Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo 2002), June 2002

[29] World Wide Web Consortium (W3C). "SOAP Specifications." Version 1.2. <http://www.w3.org/TR/soap/>

Appendix A. ACE-IMS Web Service Descriptor (WSDL)

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions            targetNamespace="urn:tss.ias.umiacs.umd.edu"            xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="urn:tss.ias.umiacs.umd.edu"                                                                      xmlns:intf="urn:tss.ias.umiacs.umd.edu"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"                              xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: #axisVersion#
Built on #today#-->

  <wsdl:message name="TSS_TokenRequestResponse">
    <wsdl:paart name="TSS_TokenRequestReturn" type="soapenc:base64Binary"/>
  </wsdl:message>
  <wsdl:message name="TSS_TokenRequestRequest">
    <wsdl:part name="reqId" type="soapenc:string"/>
  </wsdl:message>

  <wsdl:message name="TSS_DoesTokenExistRequest">
    <wsdl:part name="reqId" type="soapenc:string"/>
  </wsdl:message>
  <wsdl:message name="TSS_DoesTokenExistResponse">
    <wsdl:part name="TSS_DoesTokenExistReturn" type="xsd:boolean"/>
  </wsdl:message>

  <wsdl:message name="TSS_CompareCSIRequest">
    <wsdl:part name="time" type="xsd:long"/>
    <wsdl:part name="CSI" type="soapenc:base64Binary"/>
  </wsdl:message>
  <wsdl:message name="TSS_CompareCSIResponse">
    <wsdl:part name="TSS_CompareCSIReturn" type="xsd:boolean"/>
  </wsdl:message>

  <wsdl:message name="TSS_GetServerTimeRequest">
  </wsdl:message>
  <wsdl:message name="TSS_GetServerTimeResponse">
    <wsdl:part name="TSS_GetServerTimeReturn" type="xsd:long"/>
  </wsdl:message>

  <wsdl:message name="TSS_StampRequest">
    <wsdl:part name="in_tsreq" type="soapenc:base64Binary"/>
  </wsdl:message>
  <wsdl:message name="TSS_StampResponse">
    <wsdl:part name="TSS_StampReturn" type="soapenc:base64Binary"/>
  </wsdl:message>


  <wsdl:portType name="IAS_TimeStampingSystem">
    <wsdl:operation name="TSS_GetServerTime">
      <wsdl:input message="impl:TSS_GetServerTimeRequest" name="TSS_GetServerTimeRequest"/>
      <wsdl:output message="impl:TSS_GetServerTimeResponse" name="TSS_GetServerTimeResponse"/>
    </wsdl:operation>

    <wsdl:operation name="TSS_Stamp" parameterOrder="in_tsreq">
      <wsdl:input message="impl:TSS_StampRequest" name="TSS_StampRequest"/>
      <wsdl:output message="impl:TSS_StampResponse" name="TSS_StampResponse"/>
    </wsdl:operation>

    <wsdl:operation name="TSS_DoesTokenExist" parameterOrder="reqId">
      <wsdl:input message="impl:TSS_DoesTokenExistRequest" name="TSS_DoesTokenExistRequest"/>
      <wsdl:output message="impl:TSS_DoesTokenExistResponse" name="TSS_DoesTokenExistResponse"/>
    </wsdl:operation>

    <wsdl:operation name="TSS_TokenRequest" parameterOrder="reqId">
      <wsdl:input message="impl:TSS_TokenRequestRequest" name="TSS_TokenRequestRequest"/>
      <wsdl:output message="impl:TSS_TokenRequestResponse" name="TSS_TokenRequestResponse"/>
    </wsdl:operation>

    <wsdl:operation name="TSS_CompareCSI" parameterOrder="time CSI">
      <wsdl:input message="impl:TSS_CompareCSIRequest" name="TSS_CompareCSIRequest"/>
      <wsdl:output message="impl:TSS_CompareCSIResponse" name="TSS_CompareCSIResponse"/>
    </wsdl:operation>
  </wsdl:portType>


  <wsdl:binding name="IAS_TimeStampingSystemSoapBinding" type="impl:IAS_TimeStampingSystem">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="TSS_GetServerTime">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="TSS_GetServerTimeRequest">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:input>
      <wsdl:output name="TSS_GetServerTimeResponse">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="TSS_Stamp">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="TSS_StampRequest">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:input>
      <wsdl:output name="TSS_StampResponse">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="TSS_DoesTokenExist">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="TSS_DoesTokenExistRequest">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:input>
      <wsdl:output name="TSS_DoesTokenExistResponse">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="TSS_TokenRequest">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="TSS_TokenRequestRequest">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:input>
      <wsdl:output name="TSS_TokenRequestResponse">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="TSS_CompareCSI">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="TSS_CompareCSIRequest">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:input>
      <wsdl:output name="TSS_CompareCSIResponse">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="helloWorld">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="helloWorldRequest">
        <wsdlsoap:body   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"   namespace="urn:tss.ias.umiacs.umd.edu"
```

```
use="encoded"/>
         </wsdl:input>
         <wsdl:output name="helloWorldResponse">
            <wsdlsoap:body       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"       namespace="urn:tss.ias.umiacs.umd.edu"
use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="IAS_TimeStampingSystemService">
      <wsdl:port binding="impl:IAS_TimeStampingSystemSoapBinding" name="IAS_TimeStampingSystem">
         <wsdlsoap:address
location="http://naradev05.umiacs.umd.edu:8080/IAS_TimeStampingSystem/services/IAS_TimeStampingSystem"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```