FromNowOn Web Archive System

An Internet archive service, taking you back in time.[1]

David Moore-Pitman

Rob Radez

Adam Smith

6.033, Professor Rinard, Spring 2005

[1] DISCLAIMER: Not available for times before launch of the system, or for 99.66% of all web sites. Limited time only, so act now, order today, send no money, offer good while supplies last, two to a customer, each item sold separately, batteries not included, mileage may vary, all sales are final, allow 6 weeks for delivery, some items not available, some assembly required, some restrictions may apply, void where prohibited, except in Indiana.

Table of Contents

This paper presents FromNowOn, a distributed archive cluster system whose goal is to keep time indexed histories of web sites on the Internet. By using fault-tolerant distributed data storage, web site server-driven updates, and a publicly available request interface, our system's three pieces work together to provide the desired behavior. The distributed storage is based on quorum principles, and is built on top of individual servers which run their own DBMS software for high data fidelity, alongside replication processes which ensure proper behavior. The updates made by each web site server to FromNowOn are facilitated by a client-side utility that minimizes setup time and provides additional redundancy. The public interface to the archive uses message digest signing techniques to ensure the authenticity of the pages returned to the user. These and many other features make FromNowOn the right choice for a coordinated and distributed Internet archive.

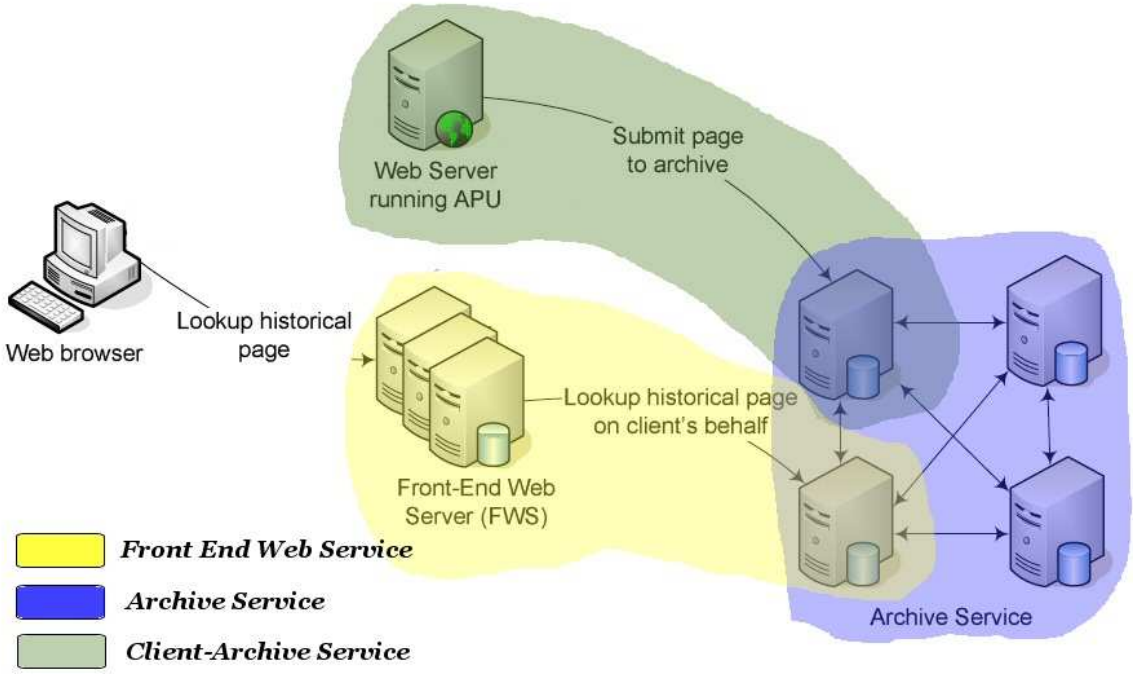
1. Introduction

The FromNowOn system has many requirements which complicate its design. The Internet is a large place, which alone sets the system complexity requirements very high. In addition to this environment, we need some layers of security, fault-tolerance, and account management.

To solve this problem, we divide the system into three pieces, called *services*. The services are called: the client-archive service, the archive service, and the front end web service, as shown below.

The job of the front end web service is to field HTTP requests from clients for historical web pages. The client-archive service is responsible for supporting the registered web site's updates, which are sent to the archive. The archive service provides functionality for authentication, account management, and generic data storage using distributed data storage techniques.

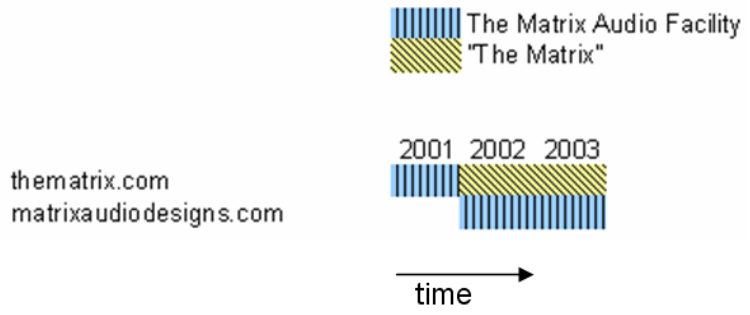
The layout of this paper is as follows. First, we will discuss the basic system paradigm which is centered around a URL-time domain. The rest of the document will outline each service; first, the client-archive service will be discussed, followed by the archive service and concluded by the front end web service.


Web Server
running APU
Submit page
to archive
Web browser
Lookup historical
page
Front-End Web
Server (FWS)
Lookup historical page
on client's behalf
Archive Service
Front End Web Service
Archive Service
Client-Archive Service

2. System Paradigm

2.1 General Paradigm

Most people model today's Internet as a collection of web sites (URLs), the design (and use) of this system requires thinking about a two-dimensional space – URLs, and time. Specifically, the content and owner of a given URL can change with time, as illustrated in the figure below, in which the ownership of thematrix.com is moved from an audio group to Warner Brothers. This broadened domain is a source system complexity.



2.2 Implications

This means that we must design our system to be robust to changing URL control with time, i.e. support deregistering and reregistering previously-owned URLs. The automatic timeout mechanism used to ensure deregistration if the previous user never deregisters is the expiration of certificates; as discussed later, if an account runs out of non-expired certificates, then it is automatically deregistered. Any site can register if it has a valid certificate provided by a root CA demonstrating control over the URL their web site is created under.

3. Client-Archive Service

3.1 Introduction

In order to achieve the goal of the entire archive system, FromNowOn needs to have some mechanism for obtaining the contents of web pages. This section will discuss a coordinated, client-driven mechanism for achieving the core system requirement, among other important features (e.g. security). This mechanism is called the client-archive service; it operates based on groups of actions (called transactions) that are formed between the client and a client utility called the APU (Archive Publishing Utility) to be sent to the archivers for processing.

The rest of this section is divided into our general approach, the requirements of the client-archive service, the service state from a static perspective, and concludes with a look at the dynamics which maintain static invariants while getting things done.

3.2 High Level Design Decision

In principle there are two approaches that could be used to retrieve periodic versions of a web site – coordinated and uncoordinated. In the first, the archive service works with each registered web server to maintain the desired behavior. In the second, the service regularly downloads each appropriate web site and saves any new data.

Uncoordinated approaches are attractive because they require no additional infrastructure on the web server side; development, coordination, and complexity costs are minimized. This is the route taken by many major archive services.

Coordinated approaches, though harder to implement, can offer many attractive benefits. The archive service can use certificates to ensure that the data is really from the domain owner. The client can notify the service of new versions of the page, so that updates are executed if and only if they are appropriate. Clients can know for sure if and when their site update was logged. Finally, the service can track the web site if it moves its Internet address.

We chose to use coordination between the client and archive service since FromNowOn's requirements call for the features provided by a coordinated service.

3.3 Service Requirements

We are given two types of requirements for this service – operations that we must support, and features that we must have. We will now detail these requirements and identify the solutions used, which are described methodically through the rest of this section.

The following operations must be supported:

- [Registering / Deregistering / Renaming] web sites
- [Adding / Deleting] certificates
- [Adding / Deleting / Updating] files

Note that all of these operations are invoked by the client on the archive service. The service is unidirectional. All operations are also initiated by the client.

The operations done on certificates are not immediately required by the design project handout; they are an indirect consequence of the security requirement. We list them in the operations requirements for completeness since the list above will set up our API discussion later.

The second class of requirements are specific features. The client-archive service must support authentication and atomicity. Authorization is necessary for us to know that operations being performed are being done by the owners of the site. As mentioned, this security is achieved by using certificates.

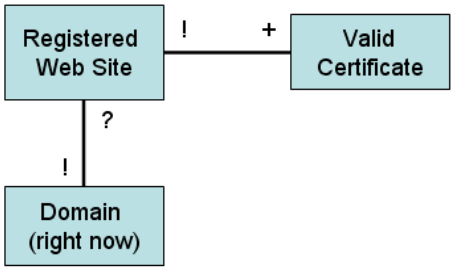
Atomicity is the assurance that operations either completely happen or the state of the system isn't affected at all. This feature ensures that the client is sure about the fate of its operations, and that the system's state does not become corrupted by incomplete transactions. The nature of that state is discussed in the following section.

3.4 Invariants – the Static Perspective

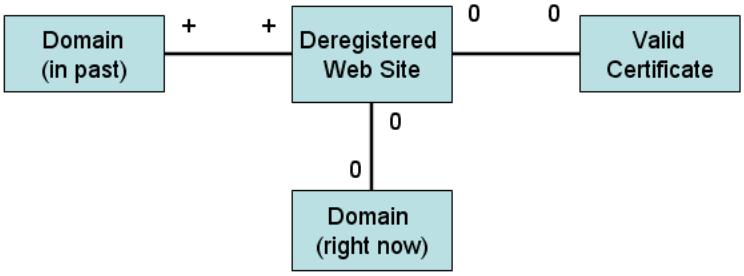
This section will describe the invariants of the client-archive service – those things that are true in any snapshot of time. After this static picture is fleshed out, the next section will describe the dynamics – the moving pieces which maintain the invariants while providing the required functionality.

At any given time each registered web site has some number of valid certificates that can be used for authentication; more than one is allowed, but there must be at least one. Valid certificates must be signed by a root CA, have not expired, and must assign authority for the domain associated with the account. Furthermore, a registered web site

must be associated with exactly one domain[2] at any given time. These relationships are illustrated in the figure below.[3]



Deregistered websites are different than registered websites. They do not have any certificates or a current domain associated with them. They have, however, had any number of domains associated with them in the past (e.g. if they moved between domains in time via a rename operation).



3.5 Dynamics

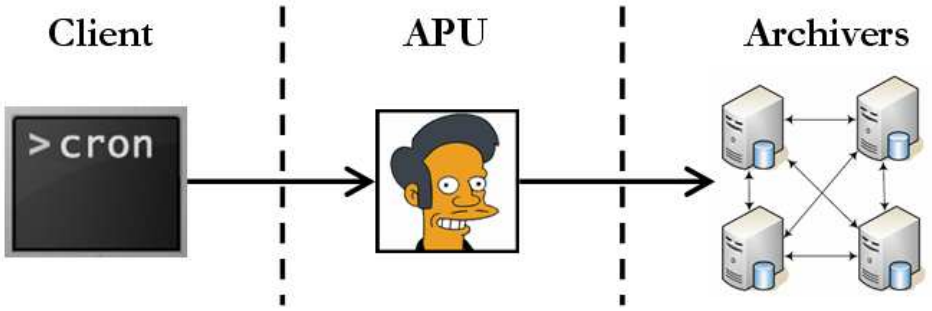
Given the invariants discussed above, this section will outline the machinery used to support the operational and feature requirements of the service. We begin by outlining the communication channel that is established between the participants, and end with the details of the communication that occurs on that channel.

Communication Channel

In this section we are interested in the communication channel that exists between the pieces of the service, namely discerning what the pieces are and the interfaces between them. At the conclusion of the discussion we will have a basis to support the necessary operations.

[2] The word "domain" here is a misnomer; when domains are referred to in this paper, it is a substitute for "parent directory." This is the top level URL at which the website begins. For example, www.geocities.com/einstein might be a different website than www.geocities.com/tigers, even though they share the same domain name.
[3] These figures use standard multiplicity notation (i.e. ? means one or zero, ! means exactly one, + means one or more, and * means zero or more) .

There are three entities in this service – a client, the APU (Archive Publishing Utility), and the archivers. The client is the software that invokes operations, such as a cron job or a publishing utility. The APU is responsible for accepting commands from the client, and passing those along to the archivers. The APU runs on the web server's machine, alongside the client. This is illustrated in the figure below.



The dashed lines represent procedure calls; the right-most one just happens to be over the Internet using some RPC tool. Thus, both interfaces have similar semantics. The APU is used as a level of indirection; it gives us flexibility in many ways. First, the client does not have to know what archive server to connect to. Second, the APU can read the appropriate files from the client's disk and format them appropriately, so that the client only needs to provide a pathname. Finally, the APU provides atomicity, which will be discussed in the next section.

The API provided by the APU for the client includes the following calls:

```
  // TRANSACTION MANAGEMENT
startTransaction(String webSite, Certificate cert)

doneTransaction()


  // FILE OPERATIONS
update(String localFilePath, String URL)  // update the
                  // file if it exists, otherwise add it

delete(String URL)


  // WEB SITE ACCOUNT OPERATIONS
register()

deregister()

addCert(Certificate newCert)

deleteCert(Certificate cert)

rename(String newWebSite)
```

None of these methods reveal that the system is supported by a network. For example, *startTransaction* is used instead of *connect*. This is an abstraction that is afforded to us by the layer of indirection that the APU provides. If somehow the system were later moved to all coincide on one machine, the semantics for the client would still be the same.

The API provided by the archivers, via RPC, is much simpler.

```
do(Transaction t)  // t contains a set of operations
                   //   and a certificate
```

In this API, a transaction is a set of operations to be performed and an authentication mechanism (i.e. a certificate).

Using the Channel in order to Get Things Done™

Now that the communication channel has been described, we will outline how that channel is used to perform operations. There are several steps to this process, including authentication, knowing which archive server to connect to, et cetera. These will be addressed in individual paragraphs; by the end of this section we will have a working model for the client-archive service.

The general idea for the use of the communication channel is the use of transactions. A transaction is a single group of operations that are performed at once. According to the APU, only one transaction can be underway at any given time for each client. Transactions are created by a call from the client to the *startTransaction* method of the APU, which also associates the transaction with a specific web site and records the authentication token to be used. Valid operations to be done are then added to the transaction one at a time. All operations may only contain either: any number of file operations OR exactly one account operation. (Where file and account operations are defined as in the APU API given above.) A transaction may also contain no operations, in which case the client can ensure that the archiver is available and its certificate is valid.

In forming a transaction, the APU does not contact any archive server until *doneTransaction* is called. When this method is called, the APU commits the transaction to storage in a local DBMS before contacting an archiver to perform the transaction. Once the transaction succeeds or fails on the archiver, the APU deletes the transaction from the local DBMS and returns the appropriate message to the client. This could take a while, depending on the size of the transaction (namely if a large file is being updated). The local DBMS ensures that if the web server (on which the APU) crashes, then the transaction will still be submitted to the archivers after the machine recovers.

Since the transaction is not sent to the archiver until it is done, a design decision must be made about when to read local files from disk. This system copies files from disk into the APU's working memory (i.e. the local DBMS transaction being built for this client-

archiver transaction) as soon as they are referenced through an *update* operation. Although this consumes additional space, which could be troublesome for very large files, it ensures that the APU does not have to deal with checking that files are still present when the client calls *doneTransaction*, or when the APU is recovering from a crash.

The APU must know how to contact the archiver, namely the IP address. The APU should connect to any archive server, randomly chosen. This functionality is obtained using round-robin DNS; i.e. the APU resolves archiver.fromnowon.org and connects to that server. This decision is made instead of using *mapAMs()* to find the corresponding archiver. Since each file might be mapped into a different archive server, and multiple files can be included in a single transaction, we cannot use *mapAMs()*. Furthermore, it is nice that we don't use *mapAMs()*, since the APU will have simpler code and will not suffer from the potential moving of our archiver IP addresses.

There are two scopes of authentication – per transaction and, for file operations, per file. Since the APU does not know what certificates are valid for the web site being updated, all transaction authentication is performed by the archivers. If the transaction authentication fails then the RPC call will return a specific value indicating this failure, which is then returned to the client. File authentication is performed by the FWS when the file is about to be displayed; files are signed by the client using the private key associated with the certificate used to upload it.

Time stamping is another function performed strictly by the archiver; the clock of the web server machine (which runs the client and APU) is never used. This is a tough design decision. Its primary motivation was to avoid clock synchronization issues; in general the system should trust the client machine as little as possible. The cost of this design decision is that time stamps could appear as late if the transaction gets to the archiver significantly after when it was formed. For example, the transaction could be committed to the APU's DBMS just before a system crash. When the system recovers, it will send the transaction to the archiver who will then assign a time stamp to each file according to the time that the client recovered. This situation is avoided by the following related design decision:

> By convention, the client should successfully publish updated pages to the archive before RTW (Release To Web).

If the client follows this convention, then the failure scenario given above is prevented; since the APU crashed, the pages were not released to web, so it makes sense to time stamp files with the recovered time. However, once again the rule applies that the system should trust the system as little as possible. The system does not have any control over the ordering between RTW and committing transactions. If the user does not follow the convention, the time stamps will be slightly off; the displacement is only slight because the web server was down during the time difference anyway.

Changing the past is not supported by the APU or the entire archive system. This limitation is imposed both for simplicity, and because it would be impossible to authenticate a user of a deregistered site who later wanted to completely remove a file.

Finally, modifications to the files being operated on are not made by the APU or anywhere else in the communication channel. All renaming of files and relative paths in HTML files is done elsewhere. This makes sense because the communication channel is only designed to transport data. Similar functions should be located at the same place in the overall system.

4. Archive Service

4.1 Introduction

The backbone of our archive is the archive service, containing two sets of servers: master servers used to store general information regarding clients, certificates, and domain registration, and archive servers, which host the contents of web sites. In both cases, we rely upon the idea of a read and write quorum to successfully utilize the distributed databases. Both types of servers use a database managing system to improve flexibility, resource costs, and organization.

4.2 Client Information Storage

To authenticate client web servers, we need to store certain information about them. This includes the domain name of the website, a unique ID for the client, any public certificates for the client, whether the client is active, and a timestamp for when the client information was last updated. To store this information, we have a cluster of five master servers running a DBMS. Client web servers connect to a hostname that uses round-robin DNS to redirect their request to any one of the five servers. The server they connect to authenticates them using the client's certificate that has been signed by the certificate authority. Once connected, the client can add a new certificate, change the domain name, or deregister itself. For those transactions to commit though, the master server contacts the other master servers to update their databases too. The transaction commits when two of the other servers in addition to the local server report that they have committed the data, for a total write quorum of three of the five servers. When the archive servers need to authenticate a client web server, they connect to a read quorum of three master servers and read the timestamp of the last modification of the client metadata and then use the most recent data to get the valid certificates and unique ID.

4.3 Archive Storage & Transactions

To archive a page, a web server uses the APU to upload that page (signed by the web server's private key) to an individual server within the archive service. The APU connects to a hostname that resolves to a random individual archive server. The archive server that the client web servers connects to is responsible for contacting a read quorum of the master servers and confirming that the client web server is authorized to insert data for the given URL. Once a web server has uploaded a version of a file using the APU, the archive server has to store that file in a fault tolerant and easily accessible manner. Initially upon receiving a file, the archive machine stores in a temporary database table the URL, the timestamp, and the data for the file (either the bytes of the file or whether it was deleted). The byte stream is

stored directly in the database. We use a DBMS for a number of reasons: it is very fast to search for different versions of files, it is very simple to store metadata for files that can be accessed without a disk read, and it takes care of caching commonly requested data without requiring additional implementation work by us. While a DBMS is a complicated program, it greatly increases the speed of the service and decreases the complexity required to implement the service. It is important to note that the databases on each archive server are independent of each other and there is no master database.

Once the web server calls *doTransaction()*, the archiver runs *mapAMs()* with each uploaded URL in the temporary database table and retrieves the IP addresses that the URL should be stored on. The archive server that the client web server connected to is not necessarily among the set of IP addresses that *mapAMs()* returns for the uploaded URL. The archive server that has been contacted by the client web server replicates data to the set of IP addresses in the same way, regardless of whether or not its own IP address is contained in that set of IP addresses. For robustness, we store five replicas of the data. Five replicas allows us to have read and write quorums of three machines that allow for two machines to be down of the five for any reason (i.e., hacker or hardware failure). The archiver then connects to the IP addresses *mapAMs()* returns, authenticating itself using a certificate that each archiver machine has and passes the URL, unique ID of the client web server provided by the client web server's authentication, timestamp, and data to each remote IP address's 'archive' table. The APU's RPC doTransaction() does not return until a write quorum of three machines has acknowledged storing the data. The commit point for the transaction is when three machines have written the data, which might not necessarily be known to the archive server trying to save replicas. If the APU or archive server fails before this commit point, then the data will not be displayed to users. If the APU or archive server fails after this commit point but before doTransaction() returns, then the data will still be available to users. If *mapAMs()* returns a set of IP addresses that do not respond to queries, then doTransaction() will return an error.

The protocol that the archive servers use to communicate amongst each other and that front-end web servers use to query the archive servers is that of a SQL client connecting to the DBMS running on each archive server.

```
//SQL COMMANDS FOR ARCHIVER

// When an APU wants to update a file in the archive, or an archive server wants to
replicate data to another archive server, the APU or server starts a SQL transaction:
        BEGIN;
       INSERT INTO archive (URL, CWSID, Timestamp, Signed_data, Deleted) {
"http://www.foo.com/bar.html", 1, 20050505110004, "...", False };
        COMMIT;

//When an APU wants to note that a file has been deleted or an archive server wants to
//replicate that fact, the APU or server starts a transaction:
        BEGIN;
    INSERT INTO archive (URL, CWSID, Timestamp, Signed_data, Deleted)
{"http://www.foo.com/bar.html", 1, 20050505110010, "", True };
        COMMIT;

//When a front-end web server wants to retrieve data for a URL most recent to the time a
//user requested:
        SELECT Signed_data FROM archive WHERE (URL =
"http://www.foo.com/bar.html" AND Timestamp <= 20050101000000) ORDER BY
```

```
Timestamp DESC LIMIT 1;
```

Using a DBMS with standard SQL allows us to extend the service in the future without needing to update our protocol significantly. It also provides a fine-grained access control list that allows us control over what can be done to the database.

4.4 Archive Security

There are two main ways that security is maintained amongst the archivers. To start with, every archive server has its own certificate that is issued by a certificate authority that identifies it as part of our archive server pool. Second, since every piece of data in the 'archive' database table is signed by the client web server that uploaded the data, the archive servers can check that signature to make sure that the data is valid. Whenever new data is inserted into an archive server's database, a method is triggered that checks the signature on the data against the certificate that we have associated with the client web server. If the signature is valid, then the data is committed to the database. If the signature is invalid, then the commit fails. Even if an archive server is cracked, it cannot actually insert invalid data into other archive servers because the cracked server will not be able to sign that invalid data as a client web server.

4.5 Fault-tolerance

In order to maintain fault tolerance, each archiver runs a background process that cycles through all URL and timestamps in its database and checks whether other IP addresses returned from *mapAMs()* for that URL contain the same timestamps and signed data. At the same time, the archiver caches remote archivers who have recently attempted to replicate information to prevent unnecessary communication. If the local archiver contains a URL and timestamp that a remote archiver does not, the local archiver replicates the data to the remote archiver. In this way, if the local archiver becomes unavailable, the data is still accessible on another server. However, if we want to increase the amount of redundancy, we can increase k on the archiver's calls to *mapAMs()* and the data is automatically replicated to the new IP addresses after a delay. While this background process might seem expensive, its priority can be set lower than servicing requests from the front-end web servers. Also, the requests that the process is making of other archive servers are very similar to the requests that the front-end web servers make and those requests are inexpensive reads of data. If each archive server stores about 100,000 URL revisions, then over a day, the background process would only have to check a little over one URL revision per second, which does not require many resources.

4.6 Server Failures

There are three likely problems that arise with archive servers. The first is that an archive server might be cracked and compromised. The second is that an archive server might stop responding to requests. The third is that an archive server might suffer catastrophic failure and lose all of its stored data.

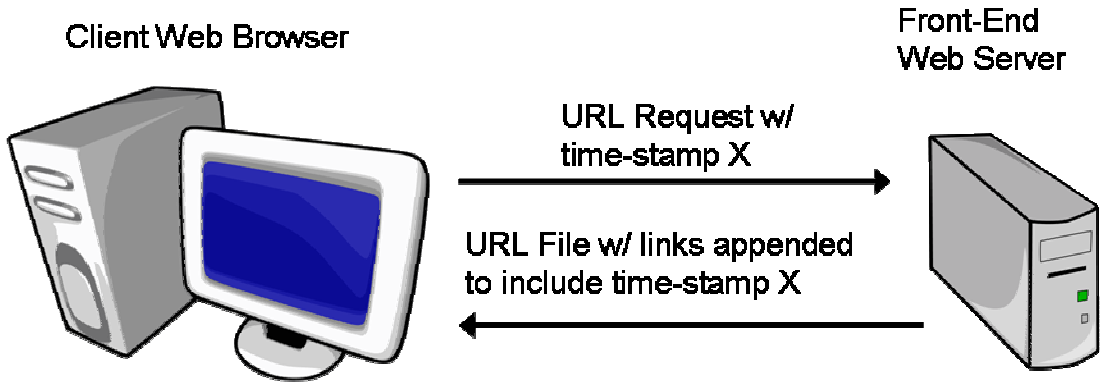
If an archive server is compromised, the certificate-based signature process allows the other archive servers to ignore any bad data that the compromised server tries to insert into the archive. Since the database access control lists do not allow updates to data already in the database, the compromised server would not be able to change any data currently in the archive either. To actually have an effect on the archive, a hacker would have to crack into at least a read quorum of machines (three machines), and even then, they would only be able to deny access to information.
If an archive server stops responding to requests, the front-end web servers would notice that that archive server has not responded to requests in a while and notify a system administrator to debug the archive server.

If an archive server suffers a catastrophic failure and loses all of the data it has stored, all the system administrators need to do is replace the hardware with a blank system. The background process running on all archive servers will eventually notice that the blank system does not have copies of the data for the URLs that the blank system is responsible for, and automatically copy that data into the blank system. The mean time to data recovery after the hardware is replaced is about 28 hours if the background processes check one URL revision per second.

1. Front End Web Service

5.1 Introduction

The Front-End Web Service (FWS) is used to process end-user web browser requests to retrieve a web page at a certain point in time. While the front-end web server is actually comprised of multiple servers, clients are serviced using round-robin DNS, which allows us to treat the connection as occurring between a single front-end web server and web browser.

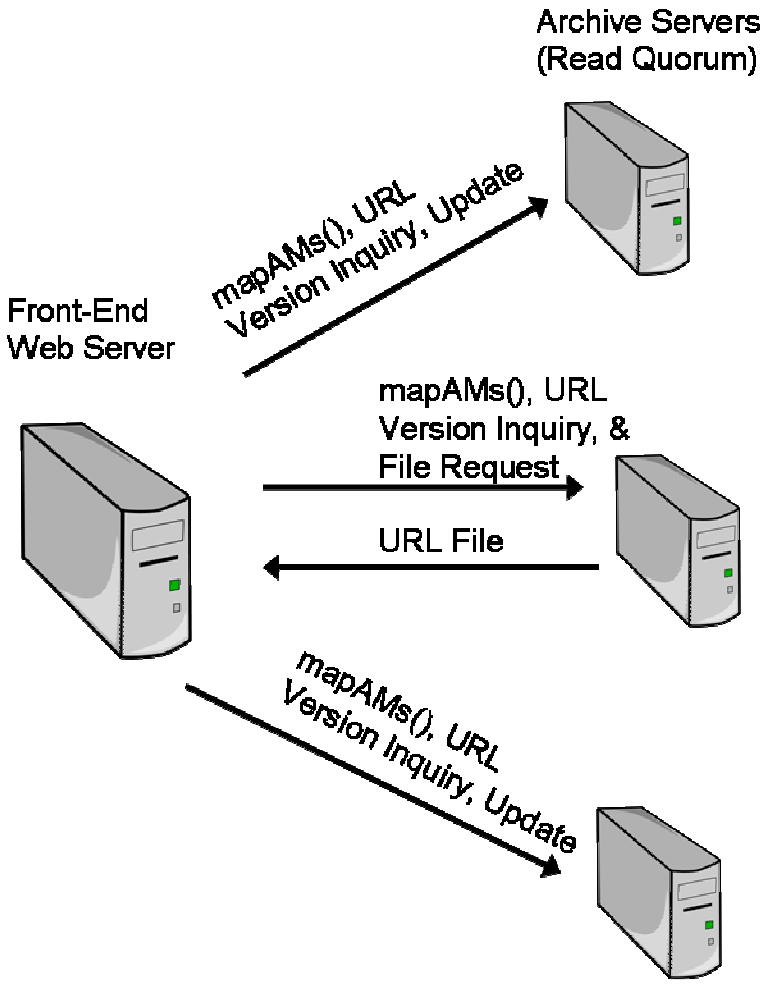


5.2 Retrieving and Serving Content

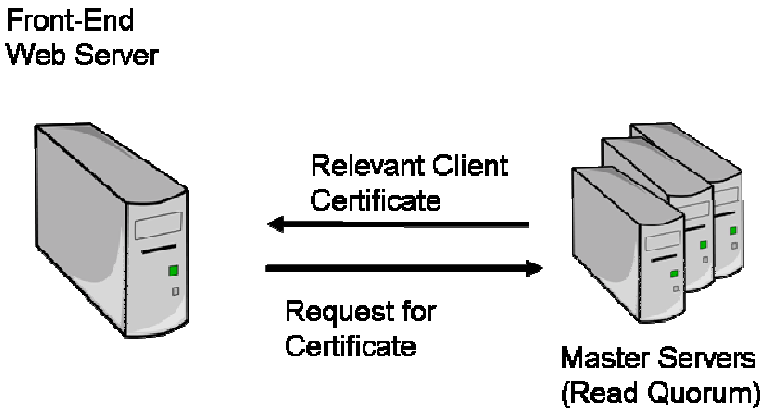
An end-user web browser requests a webpage by sending a CGI request to the FWS with the format shown in the figure below:

http://www.alltime.com/cgi/requestcontent.cgi?url=YYY&timestamp=ZZZ

The URL field of the CGI request is the absolute URL of the link, and the time stamp field represents the user requested time. If the time stamp is within 30 minutes of the current time, the FWS simply redirects the web browser to the actual website. Given the information of the URL and requested timestamp, the FWS first calls *mapAMs()*, with the supplied URL and *k* set to an appropriate number to achieve a read quorum, to obtain an array of archivers with the URL. The FWS simultaneously polls all servers in the array requesting the time of the URL closest to the supplied time stamp (regarding only URLs that come before the time stamp).



After receiving enough responses to achieve a read quorum, the FWS processes the responses to find the archiver with the URL closest to supplied time stamp (known as the *prominent* archiver). If the most recent URL version is tagged as "deleted" our front-end web server simply returns a 404 error to the user. Once the prominent archiver has been found, the FWS sends a request for the signed URL content, and retrieves the relevant web certificate from the master servers. Since the master servers are a distributed database, the FWS must again achieve a read quorum and select the most recent version of metadata from the read quorum. The FWS then checks that the content was signed using the appropriate certificate in order to guarantee that data has not been altered by a 3rd party, and removes the signature from the data. After verifying the data, links are processed as described in the algorithm below and returned to the end-user. Our security assurance to the end-user is very simple- we only serve pages we determine to be authentic. If it is determined that the page has been tampered with, an error is returned to the end-user declaring that the content has been compromised. Once the content has been served to the web browser, the FWS updates the all of the servers in the read quorum which did not contain the URL. As long as the read quorum is less than 50, we can assure that it will take less than 100 disk page reads to locate the required URL – each archive server performs one page read for locating if it has the URL, and another to retrieve the timestamp of the URL. Since our read quorum is 3, the number of page reads is significantly less than 100.




```
//FWS Content Retrieval Algorithm
retrieveContent(String URL, Int time_stamp){
//Retrieve the list of archivers in the read quorum
IPAddr[] archiver_list = mapAMs(URL, read_quorum);
if(archiver_list.size() < read_quorum){
    return readQuorumFailure; //Not enough archivers to proceed, retry then return failure
                                to web browser
}
// Returns the IP of  archivers with the URL with the closest time stamp <= time_stamp;
IPAddr[] prominent_archivers = getClosestIPToTimeStamp(archiver_list, URL,
time_stamp);

//Remove the prominent archivers from the IP list so when we return a valid response
//we can also return a list of archivers that need to be updated.
archiver_list.remove(prominent_archivers);

//Retrieve the URL and  its time stamp from a randomly selected archiver in the
// prominent archivers array
File originalContent = getURL(prominent_archivers, URL, time_stamp);
int content_time_stamp = getTimeStamp(prominent_archivers, content);

//Retrieves the relevant certificate from a read quorum of master servers
//Use the content's time stamp to retrieve the certificate used at that time to sign
//the data
Certificate cert = getRelevantCertificate(URL, content_time_stamp);
//If the content is verified…
if(contentIsCorrectlySigned(content, cert)){
    //Remove the signature to avoid sending garbage to the web browser
    content = removeSignatureFromContent(content);
    //Is it HTML? If so, process links
    if(isContentHTML(content)){
      content = changeRelativeLinksToAbsolute(content);
      content = changeLinksToFWSRequest(content);
      return content and archiver_list;
    }else{
    //It's not HTML (jpeg, mp3?) so we can just return it
    return content and archiver_list;
    }
}else{
//The content has been tampered with, return an error
return verificationError;
}
```

2. Conclusion

Our web archive service allows clients to store past versions of their website in a secure and fault-tolerant manner. We achieve this goal by dividing our service into three components: the APU service, our archive and master servers, and the front-end web server. The APU allows client web servers to connect to our archive service to modify content, and manage certificates/domain names. Our archive service is divided into two groups of servers: the master servers, which maintain metadata information about users, certificates, and domain names, and archive servers, which store the actual web site content. Archives are served to the end user's web browser through a front-end web service. At each stage we guarantee security by verifying signed content, and using certificates to identify the host sending content. Our service is fault-tolerant by replicating content across multiple archive servers. However, in order to ensure that there are not discrepancies due to propagation delays, all interactions with archive servers are performed using a read and write quorum. The APU is tolerant of failures by using the concept of reliability through atomic actions, and commit points. Essentially, by trusting no part of our system to be operating correctly, we create a robust system that confidently handles a wide array of failures and security breaches. Popular web pages are already distributed between multiple servers. If it is found that a particular web page still overloads the current read quorum, the write quorum can simply be increased to replicate the content across more servers. We do not allow clients' web servers to modify data in the past because this changes the role of the service from an archive to a dynamic storage system. Allowing this would prove to be difficult, given that as client's certificates expires; they would no longer be able to authenticate themselves as the creator of content at a given time.