FART: Fault-Tolerant Archive Retrieval Technology

Pawan Deshpande, John Lee, Lucy Mendel

6.033 Design Project 2

Liskov/Ferreira TR1 (R07)

May 5, 2005

Contents

1 Introduction **3**
- 1.1 Overview of Components . . . 3
- 1.2 Design Considerations and Assumptions . . . 5

2 Design Description **6**
- 2.1 Archive Publishing Utility (APU) . . . 6
  - 2.1.1 Subscription . . . 6
  - 2.1.2 Registration . . . 6
  - 2.1.3 Deregistration . . . 7
  - 2.1.4 DNS Renaming . . . 7
  - 2.1.5 Unsubscribing . . . 7
  - 2.1.6 Publishing . . . 7
- 2.2 Archivers . . . 10
  - 2.2.1 Archiver Filesystem . . . 10
  - 2.2.2 APU Interaction . . . 10
  - 2.2.3 APU Transaction API . . . 11
  - 2.2.4 Atomicity . . . 13
  - 2.2.5 FWS Interaction . . . 14
- 2.3 Front-End Web Servers (FWS) . . . 15
  - 2.3.1 Registration Table . . . 16
  - 2.3.2 FWS API . . . 16
  - 2.3.3 Retrieval of Archived Content . . . 19

3 Security **19**

4 Feasibility and Analysis **21**
- 4.1 APU Timestamps . . . 21
- 4.2 Archivers . . . 21
- 4.3 FWS Load . . . 22
- 4.4 Browsers and Trust . . . 22
- 4.5 Expired Domain Names . . . 23

5 Conclusion **23**

List of Figures

1 Overview of the AllTime web archive service. 4
2 Pseudo-code for the publishing routine. 9
3 Pseudo-code for `start()`. 12
4 Pseudo-code for `modify()`. 13
5 An example log for one successful transaction 14
6 A sample subset of the registration table. 16
7 Pseudo-code for `register()`. 17
8 Pseudo-code for `deregister()`. 17
9 Pseudo-code for `isRegistered()`. 18

Abstract

The AllTime, Inc. web archive service allows users to ”browse in the past” by storing old versions of web pages provided by registered clients. This paper describes a simple design that provides secure and fault-tolerant service both to users browsing the archive and clients publishing pages. The primary goal of this design is to provide availability and security of archived content. We achieve this goal by replicating data using a quorum-consensus protocol together with a checksum mechanism. While our design is susceptible to large, sudden bursts in Internet traffic, we favor our approach largely because of its simplicity and ability to tolerate failures and security issues at the cost of slightly degraded performance.

1 Introduction

The AllTime, Inc. web archive service serves two types of customers: *clients*, who are content publishers who pay AllTime to store historical versions of web pages, and *users*, who are end users browsing the archive of stored pages. Our archive service provides storage and guarantees availability of the pages of paid clients who have registered with AllTime.

We differentiate between *subscriptions* and *registrations*: a *subscriber* is a client currently paying to use the archive service. A subscriber may *register* any number of domains to allow content to be published from that domain to the archive.

1.1 Overview of Components

Our design consists of 3 components: client web servers running archive publishing utility (APU) software, front-end web servers (FWS) which handle user requests, and archivers, which store the content. The components of the system, as well as the interactions between them, are summarized in Figure 1.

The purpose of the APU is to allow registered clients to submit content to the archive. When a client notifies AllTime that content is deleted from its server, the content is not expunged

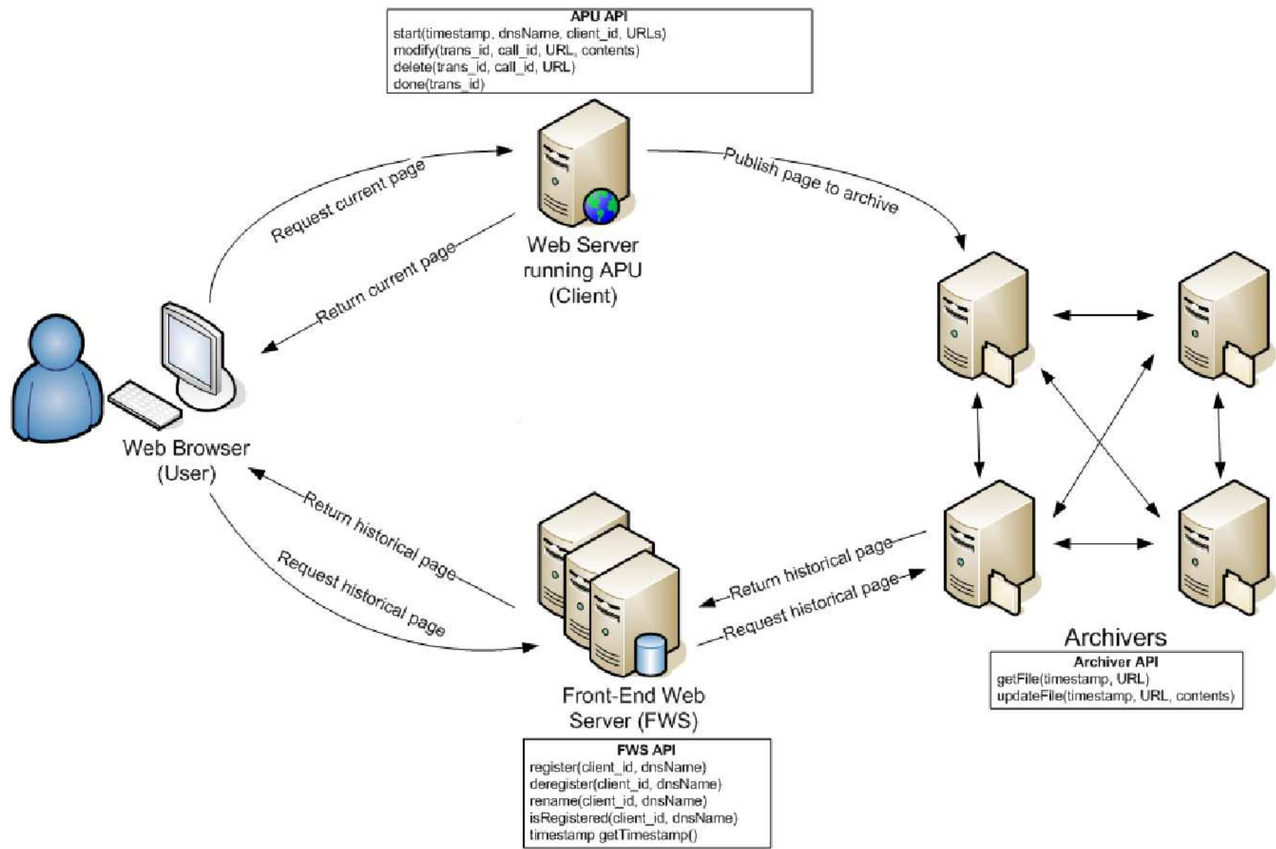

Figure 1: Overview of the AllTime web archive service.

from the archive, but instead replaced with a "Does Not Exist" (DNE) page which is returned when users make requests for a version of the page after deregistration. Therefore, as long as the client publishes all modifications to its own server in a timely manner, the archive reflects the state of the client's web site at any time.

The FWS has two primary functions: handling registration and administrative information from clients, and sending archived content to users. The administrative information is stored in the Registration Table in the FWS and is used to record when clients have registered and deregistered. The content sent to the end users is obtained by querying the archivers.

The archivers are responsible for storing the content published by the APU. The availability of a large number of archive machines allows us to create many replicas of archived content in order to provide a fault-tolerant service to clients and users.

1.2 Design Considerations and Assumptions

Availability. Our design can tolerate failures of archive machines assuming that the probability that an individual archiver fails is small. Our service handles user requests reasonably quickly for normal amounts of Internet traffic – sudden bursts in traffic to one particular page in the archive may result in slow service. Because bursts in traffic are difficult to anticipate and expensive to handle robustly, we favor a simple design where the only negative effect of sudden traffic bursts is temporarily degraded performance.

Security. Our system tolerates most attacks by malicious parties but does not cope well with repetitive denial-of-service (DoS) and other large-scale attacks. However, the system can reliably detect and cope with a compromised archiver, and is resistant to replay attacks. We assume that all communication channels between any of the components in our system are secure (using a public-key encryption scheme).

Authenticity. Our design provides end-to-end authenticity of content. We ensure that the content stored in the archive is an exact copy of the client's data. The content is then verified by the FWS, and finally sent to the user (over a secure channel, as mentioned above).

Consistency. We strive to make the content on the archivers accurately reflect the state of a client's web site. Thus, we do not allow clients to remove content from the archive. Because the client pushes data to the archivers rather than AllTime crawling the client's page, we cannot guarantee consistency. However, we assume that the client accurately publishes changes to its site in a timely manner. AllTime does not provide software that automatically detects changes – clients wishing to automate publication must use third-party tools.

2 Design Description

2.1 Archive Publishing Utility (APU)

The AllTime Publishing Utility (APU) software, which must be installed on the client's web servers, enables communication between clients and the AllTime machines. There may be a graphical user interface layered between the client and the APU API, but only the API specifications are relevant to this document. In the rest of this document, clients are assumed to interact directly with the API in order to simplify explanations.

2.1.1 Subscription

In order to use the AllTime archiving service, clients subscribe to a third party accounting service to establish payment exchanges (see Figure 1). The relevant component is that the client provides an identifier established by a certificate authority trusted by the client and by AllTime. This identifier (referred to as the client ID) is used to validate registration and deregistration commands. Subscription must only be performed once per client, and must precede all other transactions.

2.1.2 Registration

After a client is established as a subscriber to the AllTime system, the APU enables it to register a DNS name, which simply calls `FWS.register()` (see Section 2.3.2). Registering a DNS name allows a client to begin publishing to the archive using that name. Note that the registration is only accepted if the request originates from a machine with that DNS name. This is to prevent clients from simply registering an arbitrary domain name.

There is no limit on the number of web servers a client may register. Each web server should be running the APU in order to use the publishing API. All publishing transactions made by the APU include the client ID as well as the associated DNS name.

2.1.3 Deregistration

The APU deregisters DNS names by invoking `FWS.deregister()` (see Section 2.3.2), which causes browsers to receive DNE messages for requests to that DNS after the time of deregistration. However, pages are not removed from the archivers; requests for that DNS name before the time of deregistration return the appropriate pages. Unlike `FWS.register()`, we do not require deregistration requests to originate from that particular DNS name. Clients are not able to deregister arbitrary domain names, because all communication is verified with the client's certificate. Thus, given the certificate, the FWS already knows which domains are registered to which clients.

2.1.4 DNS Renaming

A client that changes DNS names can simply call `FWS.rename()`, which will atomically change the registered DNS name. The effect is equivalent to `FWS.deregister()` followed by `FWS.register()`. This API is provided by the FWS for client-side simplicity. Like `FWS.register()`, we require `FWS.rename()` to be called from a machine on the new domain.

2.1.5 Unsubscribing

Clients unsubscribe to terminate the ability to publish pages to AllTime's archivers (for example, when a client decides to stop paying for the archive service). When a client unsubscribes, all of their associated DNS names are deregistered with our system and lose the ability to modify or delete pages. However, all previously published content is retained. Thus, users can still browse a client's pages before unsubscription, but a DNE page is returned for page requests after deregistration.

2.1.6 Publishing

The APU publishes content from the web server it is running on to archivers using the following transaction API described in Section 2.2.3. The APU also relies on the `mapAMs()` command, which is hard coded into the APU software, and the `getTimestamp()` command on the FWS. The `mapAMs()` command returns the IP addresses of the archive machines hosting

the file specified by a particular URL. The k parameter to `mapAMs()` is a hard coded constant that specifies the number of IP addresses to return. The timestamp must be obtained from the FWS by the APU before starting a transaction in order to synchronize the timestamp for files replicated on different archivers. Before publication, all URL references in any page being published are replaced with references to the AllTime archive with the appropriate timestamp. Additionally, all relative links are replaced with absolute URLs. Because the APU runs on the web server hosting the content, the APU can also easily replace relative references with absolute references.

Although it is possible for clients to hack the APU software such that start uses a different timestamp than what would be returned by the FWS, the probability that paying customers would do so to their own publishing transactions is low. Because of the modularity in the archiving system, there is no way for one client to interfere with the transactions of another client. Thus, the risk is worth the guarantee that non-tampered transactions will be atomic across the entire start to done transaction. In other words, the individual modifications and deletions made within a group either all happen or else do not happen at all, assuming no more than a specified number of archiver failures (see Section 4).

A transaction is a set of modify and delete operations that must either all complete or not change anything. Each transaction is preceded by a `start()` command and ended with a `done()` command. A transaction with an individual archiver is guaranteed (by the archiver) to be atomic. When a set of modifications is ready to be made, the APU first computes a list of archivers to be contacted (using `mapAMs()` on each of the URLs and taking the union) and initiates transactions with them individually in parallel. The entire group of transactions is atomic due to the properties of the archive API and FWS. Section 2.2.3 describes how a successful `done()` on only one archiver holding a replica of a particular file guarantees that the entire transaction is successful.

Shown in Figure 2 is pseudo-code for the publishing mechanism. Here, `changes` is a list of `modify()` or `delete()` commands to be sent to the archivers.

```
publish(changes[1..n]) {
    // Get list of all archivers to be contacted by this transaction
    M = ();
    for each (change in changes[1..n])
       M = union(M, mapAMs(change.URL, k));
    end

    // Get timestamp and start() all transactions on the archivers
    Timestamp ts = FWS.getTimestamp();
    for each (m in M)
       tid[m] = m.start(ts, dnsName, client_id, changes[1..n].URL);
    end

    for each (change in changes[1..n])
       // Get a list of archivers hosting this particular file
       S = mapAMs(change.URL + ts, k); // k is a hard coded constant
       for each (s in S)
           if(change.type == MODIFY)
               s.modify(tid[s], s, change.URL, change.contents)
           else if(change.type == DELETE)
               s.delete(tid[s], s, change.URL)
       end
    end

    // Call done() on each archiver
    if all changes were successful on at least one archiver
       for each (m in M)
          m.done();
       end
    end
}
```

Figure 2: Pseudo-code for the publishing routine.

Note that the modify and delete commands are invoked only on the archivers that host the URL to be modified or deleted, whereas `start()` and `done()` are invoked by all archivers involved in any aspect of the transaction.

2.2 Archivers

In this section, we describe how the archivers process requests from the APU and FWS. The archivers are responsible for storing all of the content published by clients. Although the probability of an individual machine failure is low, failures are not infrequent considering the large number of archivers used by AllTime. Despite this, our design is resilient to a specified number of simultaneous failures dependent on the hard coded k parameter, the number of archivers storing a replica of any given file (see Section 4).

2.2.1 Archiver Filesystem

Each archiver uses a filesystem that provides a logging mechanism (such as ext3) to provide atomicity for individual write operations. The placement of files on the archivers mirrors the file structure on the client web server, except that a timestamp is appended to the filename (i.e., `http://www.cnn.com/index.html` may be stored in our filesystem as `/archive/www.cnn.com/index.html.98135013`). A slight limitation of this naming scheme is that the maximum length of a filename is decreased by a small number. However, because our service only archives static content, we expect the length of the filename to be well under the limit and disallow clients from publishing files which have extremely long names.

2.2.2 APU Interaction

Each archiver allows a client to publish pages to it. We provide atomicity at two levels: each individual method call in the API is guaranteed to be atomic, as well as each single transaction, comprised of any set of method calls. While we do not guarantee that every archiver is responsible for holding a replica of a file at all times (i.e., one archiver is down while the client tries to write the replicas), we show in Section 4 that we can tolerate a reasonable number of failures and still have a replica of the file available among a set of archivers.

The API described in Section 2.2.3 shows how atomicity is achieved at the transaction level. We stress, however, that it only guarantees that a transaction from the APU to an individual archiver is atomic. If a file is being written to k archivers, then there are k separate

transactions. However, we note that as long as at least one of the k transactions completes successfully (for each file being modified or deleted), the archive eventually will be updated by means of the quorum-consensus protocol, described in Section 2.3.3. Thus, as far as the APU is concerned, the set of k separate transactions is still atomic.

We do not allow clients to publish pages with arbitrary timestamps. We choose to place this restriction to avoid the many potential consistency issues and more accurately reflect the state of the client's web site at the time of publishing.

2.2.3 APU Transaction API

The APU file modification API consists of the following methods:

```
start(timestamp, dnsName, client_id, URLs)
modify(trans_id, call_id, URL, contents)
delete(trans_id, call_id, URL)
done(trans_id)
```

The `start()` method initiates a transaction. All files modified in this transaction are given the timestamp `timestamp`. Operations performed by this transaction may only occur on files given by the `URLs` array. First, this method verifies that the provided `client_id` and `dnsName` are registered with AllTime. If the requesting machine does not provide a valid certificate representing `client_id`, or the `client_id-dnsName` pair is not valid, then an error code is returned and the transaction is never initiated; otherwise, a unique integer (transaction ID) is returned. The transaction ID is generated simply by incrementing an internal counter. The information processed by `start()`, as well as all subsequent operations up to `done()`, are recorded in a log; Figure 5 shows an example layout of the log. The `start()` command creates a `BEGIN` record in the log. Pseudo-code for `start()` is given in Figure 3.

Each operation called after `start()` has an associated call ID (`call_id`). The purpose of these operations is to prevent a replay attack where a hacker records a `modify()` or `delete()` command and issues it repeatedly to an archiver. If each function call within a transaction

```
start(timestamp, dnsName, client_id, URLs) {
    if(isValidCertificate(client_id) AND
       APU.isRegistered(client_id, dnsName))
         trans_id = getUniqueTransactionID(); // increments some counter
         log(BEGIN, timestamp, dnsName, client_id, URLs);
         return trans_id;
    else return ERROR_CODE;
}
```

Figure 3: Pseudo-code for `start()`.

has a unique ID, then each of the methods simply ignore call IDs which have been seen already. The call ID is simply a concatenation of a unique machine identifier with an index. The unique machine identifier prevents attacks where a call is replayed on a different archiver which happens to be using the same transaction ID. The index is incremented on every call per transaction and prevents a replay attack made during a single transaction to one archiver.

The `modify()` method publishes a file. First, this method scans the log to retrieve the timestamp (`timestamp`) associated with the transaction ID (`trans_id`). It then performs a directory lookup to locate the most recent version on the machine corresponding to `URL`. If there is a version more recent than what is given by `timestamp`, we ignore this request (i.e., we do not allow clients to publish in the past). We then write contents into a new file, but with a negative timestamp (i.e., `http://a.com/b.htm` may be written to `/archive/a.com/b.htm.-91538635`). When the client calls `done()`, these timestamps will be corrected, but we negate the timestamp so that users browsing the archive will not see a new page in the middle a transaction that has not yet been completed. This method creates a `TRANS_MODIFY` record in the log. Pseudo-code for `modify()` is given in Figure 4.

Included in the `contents` argument to `modify()` is a signature consisting of a hash of the actual content (created by the client using its private key). In addition to storing this signature on disk, we also append the public key of the client and the `client_id` argument to the contents of this file. While this information is not used by the APU, it provides a security measure utilized by the FWS (described in Section 2.3).

```
modify(trans_id, call_id, URL, contents) {
    if(isValid(trans_id,call_id)) // checks for recent unique
                                  // trans_id and call_id
      timestamp = getTimestamp(trans_id);
      if(canModify(trans_id, URL)) // checks if URL was included in
                                   // the URLs argument to start()
         f = URLtoFilename(URL) + ts;
         log(TRANS_MODIFY, trans_id, call_id, URL, contents);
         writeFile(f, contents);
         return SUCCESS_CODE;
    return ERROR_CODE;
}
```

Figure 4: Pseudo-code for `modify()`.

The `delete()` method performs the exact same function as `modify()` does, except that the contents of the file created is a DNE page. In addition, the method creates a `TRANS_DELETE` record in the log.

The `done()` method updates the timestamps of all of the files associated with a particular transaction. Therefore, we scan the log for `modify()` and `delete()` records and rename the files with the negated timestamps as needed. However, before carrying out any of the filesystem rename operations, we first write a `TRANS_DONE` record, followed by a `RENAME` record for each file. When the rename operations are complete, we write a `DONE` record.

2.2.4 Atomicity

If a failure occurs immediately after `start()` is called but no operations have been performed yet, then upon recovery we simply remove the `BEGIN` record. If the failure occurs such that the archiver is repaired before the client finishes the transaction, the archiver will see that there is no `BEGIN` record for the incoming requests and will ignore them. Thus, if the archiver fails at any time during a transaction before the client calls `done()`, regardless of the length of the transaction, no modifications are made.

```
BEGIN ID=1 FILES=http://a.com/1.htm,http://a.com/2.htm DNS=a.com      \
  CLIENTID=ClientA TIMESTAMP=91259274
TRANS_MODIFY TRANS_ID=1 CALL_ID=X1 FILE=http://a.com/1.htm CONTENTS=hello
TRANS_DELETE TRANS_ID=1 CALL_ID=X2 FILE=http://a.com/2.htm
TRANS_DONE TRANS_ID=1
RENAME TRANS_ID=1 FILE=http://a.com/1.htm
RENAME TRANS_ID=1 FILE=http://a.com/2.htm
DONE TRANS_ID=1
```

Figure 5: An example log for one successful transaction on archiver `X`. Note that records for multiple simultaneous transactions may be interleaved, but we only show one transaction for simplicity in this example.

If a failure occurs during a sequence of `modify()` or `delete()` operations, we scan the log backwards to delete all temporary files associated with this transaction (i.e., the files with the negative timestamps). Then, we remove all of the records associated with this transaction from the log, so that no subsequent client commands are carried out by this particular archiver for this transaction ID. Note that all of the `RENAME` records are written at once when the archiver receives the `done()` command. Thus, if a `TRANS_DONE` record is in the log, then all of the `RENAME` records will be in the log as well because the writes the log are atomic (as mentioned above, we use a logging file system to provide this mechanism).

If a failure occurs after a client calls `done()` but before the files are finished being renamed, then we scan the log backwards for `RENAME` records for transactions where a `TRANS_DONE` record exists but a `DONE` record does not. Upon recovery, we carry out these rename operations and append the `DONE` record to the log. Note that since there is already a `TRANS_DONE` record, the client makes no more requests with this transaction ID because it has already called `done()`.

2.2.5 FWS Interaction

The FWS uses a quorum-consensus protocol (described in Section 2.3.3) to provide consistency and fault-tolerance among the many replicas of a file. The FWS only perform two types of requests to archivers:

```
getFile(timestamp, URL)
updateFile(timestamp, URL, contents)
```

The `getFile()` method performs a directory lookup for `URL`. Because filenames in our filesystem have timestamps appended to them, we search filenames for a timestamp closest to but occurring before `timestamp`. In most cases, we simply return the contents of this file back to the FWS, or an error code if no such file is found. However, before we send the content, we verify that the signature is valid using the public key that is included in the file (see Section 3). While this does not protect against many kinds of attacks, it is a quick method of dealing with a compromised server. The archiver's security failure in this particular case is hidden from the FWS; when the archiver sees such an error, it immediately stops processing all transactions and notifies an administrator that it may have been compromised. Other kinds of attacks that this scheme does not protect against are dealt with by the FWS.

The `updateFile()` method is used by the FWS when it notices that a particular archiver does not have the most recent version of a page. If the filename (appended with the timestamp) does not exist on the archiver, it is created; otherwise, it is overwritten. The archiver provides atomicity for this operation: when it starts, it creates `BEGIN`, `TRANS_MODIFY`, `RENAME`, and `DONE` operations in the log. When the FWS calls `updateFile()`, the result is exactly the same as if an APU called `start()`, `modify()`, and `done()`, except that we do not restrict timestamps to be the latest (see Section 2.2.3) since the FWS may be writing an old version that this particular archiver did not see due to a failure.

2.3 Front-End Web Servers (FWS)

The Front-End Web Servers (FWS) provide an interface to users who are browsing historical web pages by fetching content from archivers on which the requested content in stored. In our design, we assume that there are enough FWS machines that can adequately handle the volume of client requests. Furthermore, these front-end servers automatically balance load by employing Round Robin DNS Load Balancing. We also assume that FWS machines have guaranteed uptime.

2.3.1 Registration Table

The FWS maintains a table containing a list of client IDs (`CLIENT_ID`), the associated domain names (`DNS_NAME`), registration times (`REG_TIME`) and deregistration times (`DEREG_TIME`). The registration table is used by the FWS to determine if users browsing the archives should be presented with a DNE page or with content from the archive. This table is also used by the Archivers on writes to determine if the the write operation should proceed as detailed in Section 2.2.3. This table implements a relational database for fast insertions, modification and lookup operations. A subset of a sample registration table is shown in Figure 6.

CLIENT_ID	DNS_NAME	REG_TIME	DEREG_TIME
CNN	cnn.com	10684784	NULL
CNN	sports.cnn.com	10584529	NULL
CNN	cnn.com	10386739	10495029
TimeWarner	aol.com	10092351	10153952

Figure 6: A sample subset of the registration table.

In order to allow for a client to register and deregister to same domain multiple times, the table does not require that `DNS_NAME, CLIENT_ID` pairs be unique.

2.3.2 FWS API

Registration

When a client subscribes to AllTime's services, the client provides a unique client identifier as described in Section 2.1.1. For all subsequent `register()` calls from the APU, an entry in this table is added mapping the client ID to the DNS name as shown in Figure 7.

Prior to registering the new domain, the `register()` function performs two checks. First, it makes sure that the domain is not already registered by performing a lookup in the table using the `isRegistered()` function. Secondly, it must make sure that the web server registering itself is the host of the domain name that it claims to be. This is validated by performing a DNS look up on the claimed domain name, `dnsName`, and comparing the resulting IP address

```
void register(client_id, dnsName) {
    if(isRegistered(client_id, dnsName) AND
       callee.ip_addr==DNSlookup(dnsName)) {
       timestamp = getTimestamp();
       dbQuery(``INSERT INTO REGISTER_TABLE (`CLIENT_ID`   = client_id,
                                             `DNS_NAME`,   = dnsName,
                                             `DEREG_TIME`  = NULL,
                                             `REG_TIME`    = timestamp)'');
    }
}
```

Figure 7: Pseudo-code for `register()`.

with the IP address of the callee of the function. The two should match because a client web server running the APU software should have the same domain name.

If this domain is already registered, the client ID and domain name to be registered are added into the registration table. In addition, the deregistration time is set to null and the registration time is set to the current time because the domain is currently registered.

Deregistration

The `deregister()` function checks if the specified domain is already registered. If so, the deregistration time in the table is set to the current time. When the APU software deregisters a domain, it calls the deregister function of the FWS API shown in Figure 8.

```
void deregister(client_id, dnsName) {
    if(isRegistered(client_id, dnsName)) {
        timestamp = getTimestamp()
        dbQuery(``UPDATE SET `DEREG_TIME` = timestamp IN REGISTER_TABLE
                WHERE `CLIENT_ID` = client_id and `DNS_NAME` = dnsName'');
    }
}
```

Figure 8: Pseudo-code for `deregister()`.

Registration Check

The FWS API provides a method to check if a domain along with the associated client ID are currently registered. The `isRegistered()` function is called by the archivers to determine if a write operation should proceed as mentioned in Section 2.2.3

The `isRegistered()` method returns `true` if there is an entry in the table for the given client ID and a domain with the deregistration time set to `NULL`. If true, this indicates that there exists a record in the registration table that is currently registered for the specified arguments. Pseudo-code for `isRegistered()` is given in Figure 9.

```
bool isRegistered(client_id, dnsName) {
    bool isreg = dbQuery(‘‘SELECT FROM REGISTER_TABLE
                          WHERE ‘CLIENT_ID‘  = client_id
                          AND   ‘DNS_NAME‘   = dnsName
                          AND   ‘DEREG_TIME‘ = NULL’’);
    return isreg != NULL;
}
```

Figure 9: Pseudo-code for `isRegistered()`.

Timestamp

The `getTimestamp()` method returns the system time of the FWS to the caller. We also assume that all FWS machines have synchronized clocks. This function allows our entire system to standardize timestamps according to the FWS system clocks.

Rename

The `rename()` method allows a client to change their DNS name. It simply calls `register()` followed by `deregister()`. If the `register()` call fails as a result of a discrepancy between the domain name of the client machine and the domain name being registered, then the `rename()` operation is aborted by the FWS. We choose this approach so that the client does not need to make two separate calls. This method avoids the consistency problems created if we leave the APU to do two separate calls — that is, it avoids the situation where a client calls `deregister()` but crashes before calling `register()`.

2.3.3 Retrieval of Archived Content

Users browsing the archive send requests to the FWS to retrieve historical content. When the FWS receives a request for archived content, it first checks to see if the desired domain was registered at the designated time. If the site is not found in the table or if the site was not registered at the specified time, then a DNE page is returned.

However, if the site was found to be registered at the specified time, the FWS handles this request by calling `mapAMs()` with the URL as the *name* parameter and k set to the default hard coded value to retrieve the list of IP addresses of the archivers on which the desired content resides. The FWS then calls `getFile()` to each of the archivers returned by `mapAMs()`.

As described in Section 2.2.5, the archivers will then resolve which version of the page should be returned to the FWS. The FWS employs a quorum-consensus protocol to determine the correct version of the file. As a part of the quorum-consensus protocol, the archivers that return outdated content are updated by the FWS at this point using `updateFile()`. The sizes of the read and write quorums are set to be constant fractions of k and are discussed in Section 4.

The FWS then signs the page with AllTime's private key and sends it to the user who originally requested the content. Further discussion with regards to FWS's handling of security is discussed in Section 3.

3 Security

This section describes a number of scenarios involving malicious attacks and how our design copes with them.

Scenario 1. Tampering of Content on Archiver
As described in Section 2.2.5, the public key of the client is appended to the bottom of each file in the archive. On a file read operation, the archiver will attempt to authenticate the file using the public key contained within the file. If the archiver is able to successfully authenti-

cate the file, then it returns the contents to the FWS; otherwise it notifies an administrator that it has been compromised. This technique prevents basic attacks where an intruder takes over an archiver and modifies the contents of a file by checking the integrity of the contents before returning the content to other parts of the system.

Scenario 2. Replacement of Content on Archiver
A clever hacker can circumvent the security measure described in Scenario 1 by replacing the content of a file with fraudulent data signed with the hacker's private key. In addition, the hacker appends her public key to the end of this file. In this case, when the archiver reads the replaced file, it will fail to detect that the file contents have been replaced and will therefore return the replaced content to the FWS.

The FWS can prevent this tampered content from being returned to the user by checking the public keys of each of the files returned from the archivers. If the FWS detects that the files returned by the archivers have differing public keys, then the FWS resolves which key is valid for the given domain at the given time using majoritarian voting. By returning the content associated with the majority public key, the FWS can prevent false content from being returned to the end-user through masking. The FWS can also be configured to alert an administrator of a compromised machine if it notices a discrepancy in the public keys returned by the archivers.

Scenario 3. Replay Attack During Content Publishing
The publishing protocol between the client and AllTime is immune to packet tampering and spoofing attacks because all messages are encrypted with the private key of the client. To mitigate replay attacks, AllTime returns a transaction ID when the APU calls `start()`. For every call in a transaction, the client passes the transaction ID as well as a unique call ID to prevent replay attacks. While it is possible for an attacker to replay `start()` messages, the attacker's transactions will have no effect on the operation of the archivers. The countermeasures to this attack implemented in our design are described in further detail in Section 2.2.3.

Scenario 4. Spoofing of Domain Name by Client

Our design prevents a client from attempting to register a domain which it does not own by verifying that the registration request originates from a web server on the specified domain. This is implemented by a DNS lookup in the FWS `register()` method. In a similar type of attack, a client may attempt to publish content for a domain for which it is not registered. However, we are able to counter such actions by verifying that the specified domain is currently registered to the client ID associated with the publisher in the registration table (see Section 2.3.1). Furthermore, the authenticity of the client making the request is verified by the client's certificate.

4 Feasibility and Analysis

4.1 APU Timestamps

Our design allows the APU to set the timestamp associated with publication transactions. The software is made such that the same timestamp is sent to all of the archivers to which the APU publishes. While it is possible for the client to circumvent the software and have the APU send different timestamps to each archiver, we note that this is a paid service and assume that customers will not attempt to create inconsistencies in an archive of their own web site.

4.2 Archivers

The archive can tolerate a number of failures and still continue to function properly if the size of the write quorum exceeds the expected number of simultaneous failures. The size W of the write quorum is a fixed fraction of k, which is a tunable parameter. The cost of increasing k, however, is that more total disk accesses and network transactions are made per user request, which may increase the probability that an individual archiver fails. Additionally, the FWS processing a user's request is burdened with contacting a larger number of archivers, thus

requiring more processor time to determine a majority in the read quorum. For small values of k, this trade-off is worthwhile because the cost of the aforementioned operations only grows linearly with k.

If k is chosen to be 10 and the size of the write quorum is $7k/10$, then the archive can handle up to 6 simultaneous failures within that set of machines. Note, however, that this does not imply that the archive fails to function properly if more than 6 machines are down: an error will only be visible to a user if all of the failures occur within the set of machines that host replicas of the file requested by the user.

4.3 FWS Load

On every read request from a user, the FWS checks in the registration table if the specified domain name was registered at the given time. We feel that this is not significantly detrimental to the performance of read operations because the expected size of the registration table is approximately proportional to the number of registered domains, which we expect to be small.

4.4 Browsers and Trust

In order to make it easy for end users to browse the archive, we use standard HTTP transmissions so that users do not have to obtain separate archive browsing software. Browsers that support secure transmissions (using SSL) can be sure that the content is genuinely provided by AllTime. However, there is no simple way for them to verify that the content in the archive is genuinely the publisher's content. The content and its signature are provided to the user, but most web browsing software will not be able to verify a signature that was sent by AllTime but signed by another party. Thus, we provide the signature and the publisher's information with each page sent to the client, and allow users to manually verify the signature (for example, as a comment in the HTML). While this may seem to be burdensome, we choose this design because it removes the need for additional browsing software. We also note that a user must choose to trust that any separate browsing software provides valid signatures from the original publisher.

4.5 Expired Domain Names

If a client registers a domain name with AllTime and later loses ownership of it (either through expiration or some business transaction), the AllTime system will allow the client to continue publishing content for that domain name. While this is not the desired behavior, we expect this to be an infrequent occurrence which can be easily handled by an AllTime administrator.

5 Conclusion

Core goals of the AllTime, Inc. web archiving service include high fault tolerance and security, combined with a fairly simple design. Although slight degradations in performance may result while achieving these goals, we favor our approach because of its simplicity. While this simple design results in susceptibility to large, sudden bursts of Internet traffic, the overall robustness of the system provides a valuable service to commercial enterprises interested in a fault-tolerant archive retrieval technology (FART).

Acknowledgments

We would like to thank Miguel Ferreira in particular for meeting with us on several occasions to discuss our designs. Also we are deeply grateful to Professor Liskov for sharing her knowledge of quorum consensus.