Tranche:
A Distributed Repository For Immutable Files

James A. Hill*, Bryan E. Smith*, Jayson Falkner^, Philip C. Andrews*
*Departments of Biological Chemistry and Bioinformatics, University of Michigan, Ann Arbor, MI
^Single Organism Software Inc., Portland, OR
augie@umich.edu, besmit@umich.edu, jay@singleorganism.com, andrewsp@umich.edu

Abstract

We have designed and implemented the Tranche Distributed Repository (TrancheProject.org), a scalable distributed repository for the storage and dissemination of immutable files of arbitrary size. Designed and built for the biomedical research community, this system addresses the problems associated with long-term storage and distribution of files referenced in scientific publications.

The design of the system has been shaped by the needs of the research community – with integrity, provenance, persistence, availability, security, context, and citability being guiding principles. The result is a radical departure from standard models of file sharing.

Tranche was developed primarily to address the needs of researchers and journals in the proteomics community (1-2), and is currently used by ProteomeCommons.org to host proteomics and proteomics-related data sets. Tranche is configurable and is particularly useful for any data-centric research community.

In this document, we will discuss the design and development of the system, including the challenges related to the implementation and testing of the system. We will also provide relevant performance metrics.

1. Introduction

The Tranche Distributed Repository ('tränsh – French for "slice") has been designed to meet the file storage and dissemination needs of the biomedical research community – a group with highly specialized needs. Meeting these requirements has produced a unique system with distinct features and challenges.

First, Tranche references a file by its contents, not by its location. The contents of all files on the repository are hashed, and this hash is used as the token to access the file. Since a hash is based solely on the contents, it also provides a means to verify the integrity of the file. When citing a data set in a manuscript, referencing a data set by its hash means the downloader can be certain that what they are downloading was precisely what was being referenced by the publisher.

Second, the repository is a hybrid model that uses concepts from both the peer-to-peer networks and centralized servers, so it can best be described as a "distributed server model". Because we are frequently dealing with very large data sets that must be available at all times, a pure peer-to-peer model would not be feasible since nodes continuously enter and leave the network. Furthermore, a centralized model presents other issues such as scalability, and places the entire burden of hosting hardware on one group or organization. Our model accommodates a reasonable amount of individual server downtime, as well as permitting the replacement of old servers and the addition of new servers to meet growing space requirements.

Third, all files are signed by a unique X.509 certificate associated with the user who uploaded it; every file stored in the repository stores a reference to the associated certificate. This provides the file's provenance, which is a central aspect of the peer-review-centric culture of scientific research.

Fourth, files or data sets can be optionally encrypted. The uploader can set a passphrase, and the data will be encrypted. However, the uploader can then "publish" the passphrase, which will automatically decrypt the files during download. This provides researchers with pre-publication access controls with the option of leaving a data set encrypted indefinitely.

There are many other valuable features in Tranche, many of which were added due to users'

needs or our needs as administrators of the ProteomeCommons.org Tranche Repository, including versioned data sets, licensing options, and arbitrary metadata attributes to be associated with any files or data sets.

The primary deployment for Tranche is a 16-node, 80 terabyte (total disk space) repository in use by the proteomics (the study of proteins) community, available through ProteomeCommons.org. Only 5 of the 16 servers were bought by our group. The other servers have been added to the repository by other institutions in return for uploading their research data. Participating institutions have generally been quick to adopt the system – mostly because of the increased availability for their data sets.

2. Design Overview

2.1. Assumptions

While designing the repository, our design decisions were guided by these general principles:

- Hardware is of varying quality and have different features and capabilities. The server hardware will be bought and maintained by various institutions.
- Hardware failures are inevitable. Even the best hardware will fail eventually.
- Servers are distributed across multiple timezones.
- Files being deposited vary widely in size. For proteomics, some mass spectrometers output thousands of small files with one spectra each, while others output a single file containing all of the spectra.
- Deposited files should not be modified; if they are, there should be a way to demonstrate that the data has changed while making both old and new versions available.
- Servers will be housed at organizations, institutions, and universities, which have reliable, high-bandwidth Internet connections.

2.2. BigHash

The BigHash, referred to as simply a "hash" for the remainder of this document, is used to represent data on a Tranche repository and is generated by a specially-designed hashing algorithm. The hash is central to the design of the repository – it is the reference for all uploaded data. When a file is uploaded to the repository, its contents are hashed and the file is referenced by this hash. The same is true for pieces of files, known as data chunks, which we will discuss shortly. Given the data set, a hash can be regenerated, so data integrity can be verified.

The hashing function combines three different hashing algorithms with the size of the file to produce a 76-byte identifier:

- MD5 (16 bytes)
- SHA-1 (20 bytes)
- SHA-256 (32 bytes)
- Length of represented data (8 bytes)

The purpose of combining three different hashing schemes and the length of the data is to reduce the likelihood of collision. Since each hashing scheme is independently generated, the addition of each exponentially reduces the likelihood of collision.

The resulting hash is usually encoded as a base-64 string, but is also often represented using base-16 encoding, particularly when being passed in a URL. Since the identifier is 76 bytes (or 608 bits), there is a total of 2^608 possible hashes, regardless of how it is encoded.

2.3. Hash Span

A hash span is an inclusive range of hashes. This concept is central to the design of the system, and will be referenced often during this document. A "full hash span" is defined as the complete range of possible hashes – from 0 to 2^608.

2.4. File Representation

Files stored on the repository are split into data chunks, each of which is a maximum of one megabyte in size. Each file that is uploaded has an associated metadata chunk, which contains an ordered list of all the data chunks, along with other useful information. The metadata chunk is important in both locating the data chunks for any file as well as reassembling that file. For the remainder of the document, data chunks and metadata may be referred to as "chunks" when there is no need to distinguish between the two.

A directory of files can be uploaded together; these uploads are interchangeably called a "data set" or a "project". When a data set is uploaded, a special file called the ProjectFile is generated. This file is used internally by Tranche to describe the contents of the data set, and is downloaded by a client, then used to retrieve the metadata chunks for all the files and discarded from that client machine. Like any other file, the ProjectFile has a metadata chunk that is used to locate the ProjectFile data chunks and reassemble them. The hash for the ProjectFile metadata chunk is considered the hash for the data set, and is used to download the entire data set.

2.4.1. Data Chunk

Data chunks, identified by their hash values, are byte arrays with a maximum length of one megabyte. Any file uploaded to Tranche will have one or more data chunks. Data chunks contain no metainformation, and are only useful when the clent can access the associated file's metadata.

2.4.2. Metadata

Identified by the hash of the file it describes, the metadata chunk contains information used to find, describe and reconstruct the file.

The same file can be uploaded by multiple users or by the same user in multiple data sets. This is particularly likely in cases where automatically generated files are uploaded. For this reason, the metadata must distinguish between the varying contexts in which the file might appear. Together, the signature, the UNIX timestamp of upload, and the relative path of the file in a data set uniquely identifies a shared file on the repository. Since hash collision is highly unlikely, it is assumed that all metadata addressed by the same hash designate files with identical contents.

As a file is being uploaded, it might be encoded in a variety of ways, such as to encrypt or compress the data. These "file encodings" can use any supported algorithms, and can be applied to the data in any order. Each file encoding has a different hash; since there might be multiple file encodings, they must be stored in order in metadata so that they can be processed properly by the client when unencoding the data. Files are identified by their original hashes, so multiple instances of a metadata chunk may represent files that have been uploaded with different file encoding lists. For example, the same file might be uploaded three times; the first could be compressed using the LZMA algorithm, while the second could be compressed using the GZIP algorithm, and the last might not be compressed. The meta data chunk is still referenced by the same hash, which is generated from the unencoded data for the entire file. In other words, a meta data chunk's hash is independent of its file encodings.

If an encrypted file were to be referenced by the unencoded hash, in the case that another copy of the file is in a Tranche repository, one could compare the identical file's hash to discover the contents of a file. For this reason, the passphrase is hashed and this string is added to the end of the contents of the file during hashing. This serves to obscure the contents of the file while still allowing for verification of the integrity of the file.

Each file encoding has associated properties for storing arbitrary information that Tranche can used internally for important features. For example, an encrypted data set can be decrypted by setting the passphrase into the properties of the associated encryption file encoding. This means the passphrase will be publicly visible, so the same passphrase should not be used on more than one upload. Every time an encrypted file is downloaded, the download tool checks the encryption file encoding for the passphrase before requesting the user to enter one. In this way, encrypted files can be made public without performing a new upload.

2.5. ProjectFile

The ProjectFile is used to represent directory uploads, known as a "data set". This object contains the information that is necessary to reconstruct the directory during a download and is uploaded as a file to the repository after all other files have been uploaded. It is intended only for internal use, and is only stored temporarily on a client machine during a download. The hash of the ProjectFile is the hash that represents the data set.

A ProjectFile consists primarily of a collection of ProjectFile parts. A ProjectFile part consists of the relative name of the file (the directory structure plus the name of the file) and the hash of the metadata for the file. Unlike a file, the hash assigned to a data set depends on the naming of files on the uploading user's file system because a ProjectFile part contains the relative name of a file in the directory, and thus changing the name of a file or directory will change the hash of the upload.

2.6. Security

Tranche employs a federated public-key security model using X.509 certificates to authenticate requests. All authentication is handled individually by each server because there is no central authority from which to verify permissions. The servers start with a core set of trusted certificates, each given differing levels of permissions.

The possible permissions are:

- Read Configuration
- Write Data
- Write Metadata
- Delete Data
- Delete Metadata
- Write Configuration

All users have their own public certificate and private key that together represent their identity on the network. Each user's public certificate is signed by a certificate that every Tranche server on the network will recognize by default (though each server can have an individually-modified list of trusted certificates). Tranche combines the public certificate and private key into a zipped and encrypted file called the "user zip file". In practice, these files are downloaded and used by the Tranche client when a user logs in; this way, end users do not need to fully understand the security model to use the system.

Authentication is only necessary when adding, modifying or deleting data on the network or when modifying a server's configuration. The user's certificate can be signed by one of the core trusted certificates, which will set the signed certificate's permissions to the same as the signing certificate's. However, each server's configuration can add trusted certificates, so it is not required that a certificate is signed by a core certificate (or even signed at all). Signatures, which are very long hashes of the chunk bytes using the user's private key as the cryptographic cipher, are used to authenticate requests. Upon receipt of a request, a server will use the public certificate to verify that the given signature was produced by the user for the particular request.

When appropriate, nonces are used to protect against replay attacks. A nonce is a random 32-byte

value that a client requests from a server. The server stores this nonce for a short time until it is returned with the client's request, when it is verified and discarded. If a third party that is listening to a client request replays a message sent to a server, the nonce in the request would not match any available nonces and would be discarded.

When files are being uploaded to the repository, they can be encrypted with a passphrase. Tranche currently uses AES-256 to encrypt the files on the local computer before sending the pieces across the network, so at no time will any of the files be moving across the Internet while in an unencrypted form. Also, no one can reconstruct the files from the encrypted chunk without the passphrase, including the server administrators and Tranche developers.

2.7. Network Architecture

There are three different classes of agents in this architecture: clients, data servers, and routing servers. Clients do not share data with each other – therefore, this is not a peer-to-peer model. Instead, the model that the Tranche architecture employs is best described as a distributed server model.

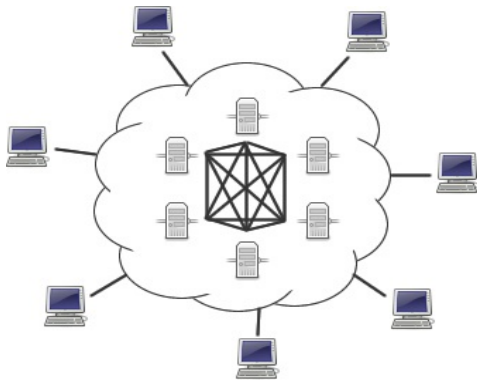

Illustration 1: Distributed Server Model

There are several reasons why the architecture was devised in this way. The classic peer-to-peer model of clients joining and leaving the network frequently does not work for data that needs to be available at all times. Because the repository will contain very large data sets, the cost of moving the data between nodes is very high. Storing the data in multiple locations allows for distributed communications, which lets the client processes parallelize requests to increase throughput. Also, using a distributed system spreads the costs and responsibilities between the providers, who are often also the most prominent users in the case of ProteomeCommons.org. This is not required; however, this participatory model has succeeded in the ProteomeCommons.org Tranche Repository. Just as it spreads the costs, a distributed system also spreads the risk of outages – the more geographically distributed the system, the less likely there will be a large portion of the data that is unavailable at any moment.

One of the primary goals of the repository is to keep a certain number of replications of all chunks available at all times. This allows for multiple failures without impacting availability. With this system, as the number of servers in the network grows, the liklihood that any fixed number of servers will be unavailable at any given time increases, but the likelihood that data will be unavailable might decrease since each server will have a smaller portion of the network's data. Data availability with the new network model is not fully modeled; however, it will provide improvements to the existing model.

2.7.1. Core Servers

Every configuration of a Tranche network must contain a set of Tranche server URLs. As long as one of these servers or the web portal (see *Interfaces*) is online, a client or server is able to get the current status of the entire network. This list serves a second purpose: it defines the set of "core" servers, which are trusted to an extent, as you will see in the remainder of this document.

The system allows for servers to join the network at any time; these are "non-core" servers. The non-core servers can be used if clients specify that they should be used, and they provide a gateway to the rest of the network. However, the data stored on them does not count towards the number of chunk replications required on the network, among other limitations.

2.7.2. Data Server

Data servers store chunks for the repository. Data servers work mostly independently of a central authority, relying on their individual configurations to guide their behavior. A data server stores only the chunks with hashes that are within the hash spans of the server plus the "sticky" files that have been designated to reside on the server.

The data server will only accept chunks with hashes that are within the target hash spans of the server. To clarify the difference between hash spans and target hash spans: hash spans encompass all of the chunks that the data server has available to read, while target hash spans encompass all of the chunks that the data server will accept. For example, during the reduction of data on the server, the target hash spans are reduced and the internal processes begin moving data off the server. During this time, some data is still stored on the server, so the hash spans reflect that.

2.7.3. Routing Server

The purpose of a routing server is to allow for a single connection to multiple servers. When a network grows very large, the number of servers that are required to maintain a connection with a full hash span of servers can grow to a prohibitive size. Assuming an average capacity of 5 terabytes per server, a network with less than 5 terabytes of data would require a client to connect to only one server for a full hash span. At almost 50 terabytes, the client would be required to connect to 10 servers for a full hash span. At 500 terabytes – 100 servers, and at 5 petabytes – about 1,000 servers, and so on. There is variability in the maximum number of TCP connections a client's system can handle because of hardware and operating systems. There is also a cost associated with establishing and maintaining each connection, so minimizing the number of required connections to connect to enough servers to access a full hash span is a primary design goal and is the reason the routing server exists.

Routing servers do not store any chunks; they are simply a special interface of a Tranche server that maintains connections with a specified set of Tranche servers. A client communicates with a routing server just as it would with a data server; it does not necessarily know that it is communicating with a routing server. As soon as the routing server receives a request, the request is forwarded to the proper server in the collection of servers to which it routes. A routing server's collection of hash spans are the hash spans of all the servers to which it routes all client requests. Because routing servers do not store any data locally, there is no need for a hard drive. The most significant bottleneck for a routing server is its bandwidth.

2.8. Time

Some key features like activity logs and user certificates reference UNIX timestamps, so all Tranche applications need to have synchronized clocks. Tranche contacts stratum 2 Network Time Protocol servers to determine local time offset with a given time zone – which one doesn't matter as long as they're all on the same one (we synchronize on the United States eastern time zone).

2.9 Sticky Data

Files can be designated to reside on a certain collection of data servers in addition to their normal replications. This can allow for local access to specific data sets and additional replications of the file. The host names of the servers on which a file should be "stuck" are stored in the metadata of the file.

2.9. Licensing

When a user uploads a data set to a repository, the user can include a license with the data set. The license file becomes part of the data set, so it therefore also becomes part of the hash that is used to refer to that data set and the data set cannot be downloaded completely without also

downloading the license.

The licenses that can be used with a Tranche repository are configurable. The ProteomeCommons.org Tranche Repository was an early adopter of the Creative Commons Zero waiver, which "is a universal waiver that may be used by anyone wishing to permanently surrender the copyright and database rights they may have in a work, thereby placing it as nearly as possible into the public domain" (3).

2.10. Versioning

There is support for versioning data sets, which is the only option for providing revisions to uploaded files. The hashes for older and newer versions of a data set are stored in the metadata, so a client can traverse different versions of the same data set using this doubly-linked list of references.

3. System Interactions

3.1. Starting Up

The first step in a Tranche process is to get the status of all the servers on the network. The network configuration files contain a list of core server URLs to which a process can connect and get the current status of the network. Each server on the network maintains a "network status table" with each row in the table representing a server on the network. The rows in the table are updated at regular intervals to ensure that its information is up-to-date. The update process is more fully described in a later section.

A row in the status table contains the following information:

- Host name – the domain name or IP address for the server (e.g. ProteomeCommons.org)
- Name – an arbitrary name for the server, as it should be presented to clients (e.g. ProteomeCommons)
- Group name – an arbitrary name for the group of servers to which the server belongs (e.g. University of Michigan)
- Port – the port to which the server is bound (e.g. 443)
- Hash span collection – the hash spans for chunks that are meant to be currently stored on the server.
- Target hash span collection – the hash spans for chunks to which the server is migrating
- Update timestamp – the UNIX timestamp denoting the last time the information in the row was updated.
- Flags
    - SSL – whether the server communicates over secure socket layers
    - Online – whether the server is online
    - Readable – whether the server allows for its data to be read
    - Writable – whether the server allows data to be written to it
    - Data store – whether the server stores data chunks on it (i.e., to differentiate between routing and data servers)

Upon obtaining the network status, the necessary connections are made for using the repository. The connections that are maintained depends on whether the the process is for a client, data server or routing server.

A client needs to maintain a connection with a collection of servers that, when combined, contain a full hash span; additionally, these connections must exclusively be to core servers that are all both readable and writable, permitting a client to download or upload any data regardless of its associated hash. Descriptions of the connections that servers need to make are available in the server sections.

3.2. Uploading

Uploading files to the repository requires the creation of a signature using a certificate that is either trusted (has permission to write) or signed by a trusted certificate on each of the servers to which it uploads. Optionally, uploads can be encrypted using a passphrase. Upon upload, files are hashed before being broken into data chunks with a maximum size of one megabyte. The hash is generated for each chunk, and the chunk is uploaded.

When uploading a chunk, the client first determines the route to all the servers that have the hash of the given chunk within their target hash spans. Because there is no way to check which servers have a replication of the chunk (a connection is maintained only with one of the *x* servers that are to be uploaded to), the chunk is uploaded without first checking whether the server already has the chunk. The payload sent consists of the chunk hash (or both the data chunk hash and the metadata chunk hash in the case of a data chunk upload), a signature, and the data. Upon receipt of the chunk, the server will validate the request before storing the chunk and sending the chunk to the other servers to which the client has requested a copy be sent. Transmission errors are detected by recalculating the hash of data chunks and by verifying that metadata can be parsed without error. While the client is awaiting a response, the server sends keep-alive signals to avoid timing out.

Nonces are not used when setting chunks, which reduces the overall latency by half. The consequences of playback are reversible and do not significantly impact the integrity of the repository.

In the case that a metadata is already online, the user's signature and upload properties must be added to the existing metadata. The upload of a file is differentiated by its UNIX timestamp of upload, uploader's user name, and the relative path of the file in a directory upload (if applicable). For this reason, the client upload process must try to download the metadata from the repository each time it is going to upload a file.

The client communicates with only one of the core servers to which it will upload the chunk. This lessens the amount of bandwidth used by the client and shifts some of the work to the server, while also lessening the number of connections required by the client. Clients upload directly to non-core servers because core servers will not communicate with non-core servers.

Upon completion of an upload, the hash is returned for the file uploaded or for the ProjectFile that represents the directory uploaded.

3.3. Downloading

Downloading from the repository does not require authentication – anybody can download any public or encrypted (with a passphrase) data set from a Tranche repository.

The download process takes as input a hash that is associated with a metadata. Upon initiation, the download tool will download the metadata, which are never encrypted, so the passphrase for encrypted data is not required until attempting to download a file's contents.

The download process can skip downloading files that already exist in the desired location on the local file system. For each file it finds, it hashes the file and compares the hash against the hash of the file that is about to be downloaded. If the hash does not match, the process will overwrite the existing file.

The expected server location of the data chunks on the network is determined by the hash spans assigned to each server. If a server does not have a chunk with a hash within its hash spans, it will try to get the chunk from the core servers that also have that hash within their hash spans. If found on one of the other core servers, the first server will store that chunk locally if possible before sending the chunk to the requesting client, thus creating another replication of the chunk.

Validation of the download protects against man-in-the-middle attacks. This validation consists of hashing the data chunks as they are downloaded and comparing the resulting hash against the expected hash, then doing the same for each file as they are downloaded and unencoded.

For the special circumstance of many small files in a data set, batching the chunk requests together significantly reduces the download time by reducing the latency.

3.4. Deleting

Only certificates with delete permissions can perform deletes; the uploader of a data set does not automatically have delete permissions. This also means that any user with permission to delete is capable of deleting any other user's data sets. For this reason, delete privileges are limited to administrators and automated administrative processes.

To fully delete a data set from the repository, every replication of every chunk must be deleted. Deletes are not commonly performed, as data sets that are uploaded are generally expected to remain online and revisions can be made through versioning uploads. While the bandwidth required for a deletion is negligible, the latency is the largest time factor. For this reason, batching requests are used extensively.

Both data chunks and metadata chunks may be shared between uploaded files. For this reason, delete requests must specify which file is to be downloaded. When deleting a metadata chunk, the request requires the user name, UNIX timestamp of upload, and relative path in directory as parameters for deletion to differentiate between file uploads. A deletion of a data chunk requires both the hash of the data chunk as well as the hash of the file to which it belongs.

4. Data Server

4.1. Activity Log

Data servers log the activities that impact their data store (see *Data Storage*). Each activity is logged with the following attributes:

- UNIX Timestamp
- Type (e.g. delete data)
- Hash
- Signature
- Other Attributes

"Other Attributes" refers to activity attributes that are specific to certain types of activities. For example, the "delete data chunk" activity takes as arguments both the hash of the data chunk and the hash of the metadata chunk to which it belongs.

Because permissions are federated among servers, the signatures for each activity are recorded to allow for later validation by other servers, which may have different permissions for different certificates or different trusted certificates completely.

The activity log is stored in the following separate random access files:

- Activity log file: fixed-length entries containing timestamp, action type, hash, signatures key and attributes key
- Signatures index file: fixed-length entries containing a key, offset and length
- Signatures file: variable-length entries only containing signature
- Other attributes index file: fixed-length entries containing a key, offset and length
- Other attributes file: variable-length entries only containing any additional attributes

The activities are logged chronologically with fixed-length entries, and the time to search a log with n entries by timestamp is $O(\log n)$. Searching for an entry without knowing the timestamp requires a linear search, though there is no current use case that requires this.

When an activity log file entry is accessed, the associated signatures key is used to get the signature offset and length in the signatures index file, which is another $O(\log n)$ search. With that

information, we read in the signature from the signatures file in *O(1)*.

Each activity log entry is approximately 100 bytes. Assuming the average data chunk size is 400 kilobytes and half the chunks are metadata of negligible size, the average chunk is approximately 200 kilobytes. This means the storage requirements for the activity logs is about 0.05%,or 1/2,000 of the total storage.

4.2. Starting Up

Upon getting the status of the network, the server establishes connections with all the servers with which is shares an overlapping hash span plus the servers from which it needs to perform network status updates plus a full hash span of servers, overlapping when possible.

To protect changes made to the network, servers must go through a process of verifying their data store before they share it with the rest of the network. To understand why, consider a server with a copy of a chunk that is offline. A user might delete that chunk from the rest of the network, but the copy that is on the offline server will still be intact. When that server comes back online, its copy will become available, and other servers will begin to replicate it, salvaging the deleted chunk.

To solve this problem, a server in the process of starting up becomes write-only so it can record changes while it verifies its data store; while performing this step, its data store is not available to the rest of the network. It will then query each server for all their logged activities between the time it went offline (which is known based on its own activity log) and the current timestamp. Chunks are deleted if any logged deletions are found and their associated signatures belong to trusted users, then the server removes the write-only restriction and downloads newly added chunks.

The last step in the process is notifying other servers on the network that this server exists and is online. This is done during the update network status table process, so it will not happen until *update interval* time has passed.

4.3. Configuration

There are several aspects to the configuration of a data server: trusted users, hash spans, target hash spans, data directories, server configurations, and properties. The configuration is also the object through which performance statistics and status are reported.

The list of trusted users is part of the system's security model. For each user, their public certificate is stored and permissions are defined.

Each data server can be assigned a limited number of hash spans and target hash spans. The number of hash spans is limited because hash spans are incorporated into each row of the network status table; because the network status is kept in memory, the system must protect against arbitrary memory consumption, especially considering the non-core servers that might participate. These hash spans essentially define how much (as well as which) data is stored on the server. Generally, the percentage of a full hash span the server's hash spans take up is the percentage of the network that will be stored on the server. So if the hash spans equal half of a full hash span and there are 6 terabytes of data on the network, then the server should be expected to hold 3 terabytes.

A list of data directories define where the server stores chunks and the maximum amount of the data that can be stored in that directory. Additional directories can later be added, and existing directories can be removed. This allows individual servers to scale appropriately.

Server configurations also store the parameters for the various forms of activity that it can perform. For example, some limited interactions with the server can be done via HTTP. The server's properties store the name-value pairs for variables that define the behavior of the server.

4.4. Data Storage

Chunks are stored in a b-tree structure; insertions, searches and deletions run in logarithmic time, O(log n). A leaf node in the structure is called a "data block". The b-tree begins with a single layer

of 256 data blocks. In this top-level data block, all the chunks have hashes which, when base-16 encoded, share the first two characters. These data blocks are hence identified by these two shared base-16 characters. When a data block contains 1,000 chunks or grows to 100 megabytes, it branches into a new layer of 256 data blocks, each of which has the same aforementioned restrictions on size. The second layer of data blocks is named by the second two characters of the base-16 value of the chunks that are stored within them, since these children data blocks will contain chunks with base-16 encoded hashes that share the same first four characters. A data block at a depth *n* in the b-tree, where n is 1 for top-level nodes, will contain chunks with base-16 encoded hashes sharing the first 2*n characters. The data block search time in a tree with *n* data chunks is $log_{256}\ n$.

A data block is made up of a header and a body. The header is a hash table of Tranche hashes and offsets where the location of the Tranche hash entry is determined by the hashcode of the string value of the Tranche hash. When a chunk is added to the block, the data is appended to the body and the lookup information is added to the header.

The b-tree is distributed across the server's data directories. Each server can have multiple data directories (typically on separate disks) and may hold millions of chunks, so balancing the storage between all the data directories is important for balancing the acccess between the disks, reducing the likelihood of a bottleneck at the disk.

A data chunk may belong to multiple files, so the repository must keep track of which data chunks belong to which files. This information is stored in a file called a "map block" that maps the one-to-many relationship between data chunk hashes and metadata chunk hashes. There is a map block for each of the data blocks.

A map block consists of a header and a body. Like the data block, the header is a hash table of Tranche hashes and offsets. The body, unlike in the data block, consists of hashes and offsets that are used to create linked lists.

4.5. Processes

Since the network status table must be updated regularly for the proper performance of a server, this process is performed continuously. Other server processes, however, are only performed when the server is not occupied performing client requests.

4.5.1. Update Network Status Table

Updating the network status table is done in a propagation scheme because of the limited number of servers to which a server is connected. The primary goals is to minimize the staleness of the information in the status table while also minimizing the number of connections required.

Within a network status table, all rows are ordered alphabetically by host name. Starting at the local host, the rows are split into groups, and a server from each of the groups (which must be an online, non-local core server) is contacted at regular intervals for the updated status of the servers in that group.

When updating the network status table, groups are more appropriately thought of as "ranges" because the addition of servers to the network must be incorporated within the existing groups.When a request for a portion of the network status table is given, the parameters for the request are simply the host names for the end

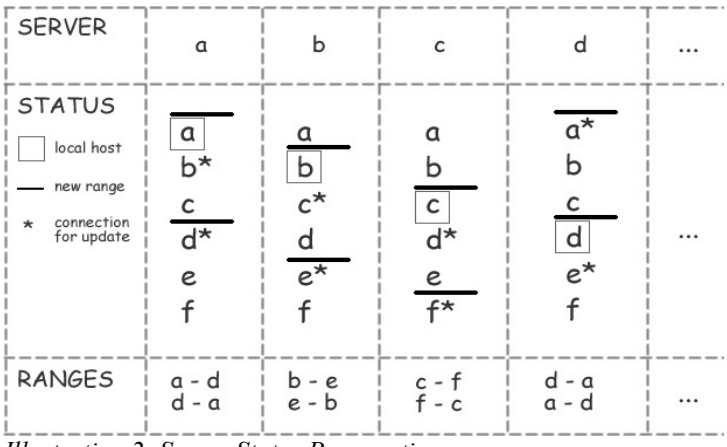

Illustration 2: Server Status Propagation

points of the ranges. The first host name in the range is inclusive, while the last host name in the range is exclusive.

It would be helpful to examine a simple scenario. Take a Tranche network with six servers total in which all are online, a group size of 3, and an update interval of 30 seconds. In the diagram, you can see that the servers have been given the labels *a* through *f* and they are sorted in alphabetical order.

In this case, server *a* updates from server *b*, which updates from server *c*, and so on. Server *a* also updates from server *d*, which updates from server *e*, etc. Modifying the variables used to determine the range size and update frequency allows for the optimization of the propagation scheme.

For server *a*, the stalest status will be for server *f*, which will be at most 90 seconds old. In general, the stalest possible status information can be calculated by *group size * update interval*.

When a server is no longer responding, it is flagged as offline and the next available server in the group is used. If no servers in the group are online, then no updates will be made until one of the servers comes back online.

Because non-core servers cannot be trusted to provide the status of other servers, they must only be asked of their own status. The task of updating from non-core servers is given to the core server that is located alphabetically before the non-core server in the status table.

To ensure that the network is notified that the local server is running, a newly-started server registers with each server which it will use for performing status updates.

4.5.2. Balance Data Directories

On occasion, a server will check the distribution of the data store's b-tree across the data directories. Data directories may be of varying sizes, so the process compares them based on their percentage used. When the percentage of used space differs by a configurable amount, the server will transfer data blocks to other data directories to lessen the disparity to an acceptable level.

4.5.3. Download Hash Span Chunks

This process requests lists of chunks that are within the server's set of hash spans from other core servers with overlapping hash spans. As it finds chunks that it does not have, it will download and store them. Although this is quite expensive in terms of bandwidth, this is the only process by which data sets that are not accessed often can have missing replications replaced.

4.5.4. Delete Superfluous Chunks

This process deletes chunks that are not within the target hash spans of the server. There are two reasons why these chunks may be residing on a server to which they do not belong: the target hash spans may have changed or the data chunks may have been associated with a sticky file. Before deleting a chunk, the server must verify whether it is part of a sticky file.

4.5.5. Download Sticky Data

Another process that runs in the background is the download of missing sticky data chunks. This process will examine the set of sticky metadata on the server, then make sure that all the data chunks listed within that metadata are on the server.

4.5.6. Notify Servers of Sticky Metadata

As the server reviews a metadata's sticky servers list, it must verify that the servers to which the metadata should be "stuck" have a copy. For security purposes, only core servers can send

notifications of sticky metadata; core servers will ignore notifications sent by non-core servers. After the metadata is stored on the appropriate server, the Download Sticky Data process will download the data chunks associated with that metadata.

4.5.7 Update Network Configuration

At set intervals, servers update their network configurations from the central authority. Network configuration parameters include core servers, certificates, and behavior variables. The central authority is usually the web portal, from which administrators can quickly make changes to the configuration for every server in the distributed repository.

5. Interface

5.1. Java API

Tranche is free, open source and available under the Apache 2.0 License (http://www.apache.org/licenses/LICENSE-2.0.html). The core software package was written in Java 5. Configuration files and core certificates define a Tranche repository. Most runnable classes take as their first argument the location of the main Tranche configuration file, which can be in the JVM, the local file system, or on the Internet.

5.2. GUI

The graphical user interface was made using the Java Swing library and is opened using Java Web Start. The Advanced GUI has extended functionality, but for most users, the only times the GUI must be launched are when uploading and downloading. Most other other management activities can be performed more easily through the web portal.

5.3. Web Portal

While a generic web portal for interaction with a Tranche repository has not yet been created, ProteomeCommons.org is a prototype for this type of application. The web portal is the central user access point for the repository, where users can search through the contents of the repository and manage their uploaded data. This is analogous to parts of a BitTorrent tracker and index website.

The web portal controls the assignment of certificates. For ProteomeCommons.org, users must sign up and go through a simple vetting process before being allowed to upload to the repository. When signing in through the GUI, a request is sent to ProteomeCommons.org, then a certificate is sent back. This allowed users that were uncomfortable with the certificate system to easily use Tranche.

Tranche Annotations was developed to contextualize the data sets in an accompanying Tranche Distributed Repository. The core feature of this software is that it allows for a dynamic information model, which allows metadata stored in the database to evolve along with the field as new technologies, new terms, and new data relationships are developed. This reflects the reality of basic scientific research where new technologies and methods lead to new terms, new concepts, and new relationships between terms. Consistent with the design of the Tranche Distributed Repository, the annotation system was designed to be configurable, and because it works through a dynamic information model, it can be applied to many other fields. The metadata is stored in a MySQL database and is managed through the web portal.

6. Measurements

6.1. Controlled Measurements

A repository was set up between three identical commodity desktop computers connected on a LAN. They all contain dual-core AMD Athlon 64-bit 3600+ processors, 4 GB of DDR2 RAM, a 7,200 RPM SATA hard drive, an NVIDIA Gigabit LAN controller, and run the Ubuntu 8.10 operating system. They are connected by a Netgear 8-port gigabit switch. A fourth identical computer connected on the same switch was used to take the measurements.

Measuring the storage capacity and usage of a Tranche repository can be tricky. Because data is stored and referenced by contents, files may overlap, so identical files or identical parts of files willd not take any more space. Additionally, uploads can be compressed upon upload, and certain types of file (typically plain-text) are often effectively compressed. For example, mass spectrometry-based proteomics data can be highly repetitive (depending on the file format), sometimes leading to compression rates of 80% or higher.

The three data sets that were used to take measurements were downloaded from the ProteomeCommons.org Tranche Repository, so they represent a range of real-world data sets. The data sets differ primarily in the number and size of files they contain.

Data Set	Files	Size
A	11,196	8.2 GB
B	10	1.1 GB
C	1	1.3 GB

Table 1: Test Data Sets

Data Set	Compressed	Time Up (s)	Avg Speed (MB/s)	Time Down (s)	Avg Speed (MB/s)
A	TRUE	4284	1.96	1022	8.22
A	FALSE	4362	1.92	978	8.59
B	TRUE	257	4.38	127	8.87
B	FALSE	319	3.53	125	9.01
C	TRUE	447	2.98	300	4.44
C	FALSE	381	3.49	218	6.11

Table 2: Upload and Download Rates

6.2. Real World Measurements

[TAKE MEASUREMENTS SOON AFTER PUSH]

7. Experiences

Many of the design decisions took some trial-and-error experimentation to reach, and they will undoubtedly be continually refined. For example, all standard user interaction with a Tranche repository was once performed through the GUI. Over time, we have received a lot of valuable experience and feedback concerning the performance and usability of the graphical application; some users found it to be lacking in speed, usability, and familiarity. We used the feedback to develope the revised ProteomeCommons.org, which offers management tasks as well as a conventional web interface for searching and browsing data sets, among other community-oriented capabilities.

At least of of our design improvements occurred because of an accident that happened after

setting up a group of four computers to run a process counting the number of chunk replications for each data set. With about 20 million chunks and about 12 servers on the network at the time, this process was expected to take a few weeks to finish. A couple of days into the process, we received an urgent email from the medical center IT at the University of Michigan instructing us to immediately turn off the processes – the requests were causing a denial of service to some key university systems (like voicemail). We still needed to run the process, so we developed a solution: to batch together many small requests into a single packet. After comparing the average latency time to the processing time of a simple request, it became clear that batching was a solution that could be applied to Tranche in general.

Tranche underwent a significant architectural redesign that began in the summer of 2009. Before the redesign, all clients and servers had to connect to every server on the network, and updated information about each server's status by regularly pinging it as well as requesting its configuration. Also prior to the redesign, hash spans were only suggestions, which meant that chunk location was non-deterministic; a process would need to check every server before determining a chunk was not in the repository. When presented with a use-case for several petabytes of data, it was clear that the system would not be able to scale. The network status problem was addressed by making a serializable table where each of the servers was represented by a row in the table, then designing propagation rules to notify the network of a server's connectivity and configuration changes. The work of updating the network status was cut to a small fraction of its previous traffic and clients could get the status of the entire network in a single request. The chunk location problem was addressed by changing the system design to require a deterministic location for chunks based on hash spans, so as the scale of the repository grows exponentially, the time to download does not increase at all.

8. Conclusions

The Tranche Distributed Repository has demonstrated its usefulness to the proteomics community in a short amount of time. Our design could potentially meet the needs of any data-centric community by providing a highly scalable storage and dissemination system for immutable files that can be run on commodity hardware and thus at a cost-effective price. Replicating files and distributing them across many servers speeds up downloads through parallelization while also ensuring the availability of the data in the event of a server failure. Hashing the contents of the repository ensures the integrity of the files is maintained, while vetting and tracking uploaders ensures the provenance of the data; combined, these ensure that downloaders know who uploaded the data and can verify the integrity of the data. The decentralized structure ensures against bottlenecks and provides ease-of-scalability. Since Tranche is decentralized, it can scale more easily than a centralized model, and helps prevent performance bottlenecks by distributing the load among available servers. We believe that Tranche is a promising solution to the need to store and disseminate large, immutable files.

Acknowledgements

Funding for Tranche and ProteomeCommons.org comes from the National Cancer Institute's Clinical Proteomics Technologies for Cancer and the National Center for Research Resources (NCRR award P41-RR018627).

Thank you to those that have contributed to the source code for Tranche: Jerod Falkner, Brian Maso, Panagiotis Papoulias, Eric Vander Weele, Todd Yocum.

References

[1] Bruno Domon and Ruedi Aebersold. Challenges and Opportunities In Proteomics Data Analysis. *Molecular and Cellular Proteomics*, 5: 1921-1926, 2006.

[2] Catherine A Ball, Gavin Sherlock, and Alvis Brazma. Funding High-Throughput Data

Sharing. *Nature Biotechnology*, 22: 1179-1183, 2004.

[3] 2009 March – Creative Commons. http://creativecommons.org/weblog/2009/3/page/5 [accessed 15 July 2009]