Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service

Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes

HP Laboratories

{kave.eshghi,mark.lillibridge,lawrence.wilcock,guillaume.belrose,rycharde.hawkes}

@hp.com

Abstract

We have developed a new storage system called the Jumbo Store (JS) based on encoding directory tree snapshots as graphs called HDAGs whose nodes are small variable-length chunks of data and whose edges are hash pointers. We store or transmit each node only once and encode using landmark-based chunking plus some new tricks. This leads to very efficient incremental upload and storage of successive snapshots: we report compression factors over 16x for real data; a comparison shows that our incremental upload sends only 1/5 as much data as Rsync.

To demonstrate the utility of the Jumbo Store, we have integrated it into HP Labs' prototype Utility Rendering Service (URS), which accepts rendering data in the form of directory tree snapshots from small teams of animators, renders one or more requested frames using a processor farm, and then makes the rendered frames available for download. Efficient incremental upload is crucial to the URS's usability and responsiveness because of the teams' slow Internet connections. We report on the JS's performance during a major field test of the URS where the URS was offered to 11 groups of animators for 10 months during an animation showcase to create high-quality short animations.

1 Introduction

Utility Computing describes the notion that computing resources can be offered over the Internet on a commodity basis by large providers, and purchased on-demand as required, rather like gas, electricity, or water. The widespread belief is that computation services can be offered to end users at lower cost because of the economies of scale of the provider, and because end users pay only for the resources used at any moment in time.

Utility services are utility computing systems that offer the functionality of one or more software applications rather than raw processing or storage resources. Possible utility services include finite element analysis, data mining, geological modeling, protein folding, and animation rendering. An important class of utility service, which we call batch services, primarily processes batch jobs where each job involves performing a well-defined set of computations on supplied data then returning the results of the computations. The data for a job may be large and complicated, consisting of many files carefully arranged in a file hierarchy—the animation models for rendering a movie short can require gigabytes of data and thousands of files.

Providing batch services to individual consumers or small and medium businesses under these circumstances is difficult because the slow Internet connections typical of these users make moving large amounts of data to the servers very time-consuming: uploading the animation models for a movie short over a typical ADSL line with 256 Kbits/s maximum upload bandwidth can take over 17 hours. (Downloading of results is usually less problematic because these connections offer much greater download bandwidths.)

We believe this problem can be solved in practice for many batch services if incremental uploading can be used since new jobs often use data only slightly different from previous jobs. For example, movie development, like computer program development, involves testing a series of successive animation models, each building on the previous one. To spare users the difficult and error-prone process of selecting which files need to be uploaded, the incremental uploading process needs to be automatic.

We have developed a new storage system, the Jumbo Store (JS), that stores Hash-Based Directed Acyclic Graphs (HDAGs). Unlike normal graphs, HDAG nodes refer to other nodes by their hash rather than by their location in memory. HDAGs are a generalization of Merkle trees [20] where each node is stored only once but may have multiple parents. Filesystem snapshots are stored on a Jumbo Store server by encoding them as a giant HDAG wherein each directory and file is represented by a node and each file's contents is encoded as a series of variable-size chunk nodes produced by landmark-based chunking (cf. LBFS [21]). Because each node is stored only once, stored snapshots are automatically highly compressed as redundancy both within and across snapshots is eliminated.

The Jumbo Store provides a very efficient form of incremental upload: the HDAG of the new snapshot is generated on the client and only the nodes the server does not already have are sent; the presence of nodes on the server is determined by querying by node hash. By taking advantage of the properties of HDAGs, we can do substantially less than one query per node. We show that the JS incremental upload facility is substantially faster than its obvious alternative, Rsync [26], for movie animation models.

As well as being fast, the upload protocol requires no client state and is fault tolerant: errors are detected and corrected, and a restarted upload following a client crash will not start from scratch, but make use of the portions of the directory tree that have already been transmitted. The protocol also provides very strong guarantees of correctness and completeness when it finishes.

To demonstrate the utility of the Jumbo Store, we have integrated it into a prototype Utility Rendering Service (URS) [17] developed by HP Labs, which performs the complex calculations required to create a 3D animated movie. The URS is a batch service which accepts rendering data in the form of directory tree snapshots from small teams of animators, renders one or more requested frames using a processor farm, and then makes the rendered frames available for download.

The URS research team involved over 30 people, including developers and quality assurance specialists. It is designed for use by real users and so has to be user friendly and easy to integrate into the customer computing infrastructure, with a high level of security, quality of service, and availability. To provide performance and security isolation, one instance of the URS is run for each animator team. Each URS instance uses one JS server to store that team's uploaded animation model snapshots. Each service instance may have multiple snapshots, allowing animator teams to have multiple jobs running or scheduled at the same time. Because of JS's storage compression, we can allow a large number of snapshots inexpensively.

To test the URS, it was deployed for each of 11 small teams of animators as part of an animation showcase called SE3D ("seed") [27], which ran for a period of 10 months. The URS gave the animators access to a large pool of computing resources, allowing them to create high quality animated movie shorts. The system was highly instrumented and the participants were interviewed before and afterwards. We report extensively in the second half of this paper on the JS's excellent performance during SE3D. As far as we know, this trial is the only substantial test of incremental upload for utility services.

The remainder of this paper is organized as follows: in the next section we describe the design and implementation of the Jumbo Store. In Section 3, we briefly describe the URS and how it uses the JS. In Section 4, we describe the results of the SE3D trial. In Section 5, we compare JS to Rsync using data from SE3D. In Section 6, we discuss the SE3D and Rsync comparison results. Finally, in the remaining sections we discuss related work (Section 7), future work (Section 8), and our conclusions (Section 9).

2 The Jumbo Store

The Jumbo Store (JS) is our new storage system, which stores named HDAGs—immutable data structures for representing hierarchical data—called *versions*. The JS is accessed via special JS clients. Although HDAGs can hold almost any kind of hierarchical data, we currently only provide a client that encodes snapshots of directory trees as HDAGs. This client allows uploading new snapshots of the machine it is running on, downloading existing snapshots to that machine, as well as other operations like listing and deleting versions. Figure 1 below shows the typical configuration used for incremental upload. A version can be created from the (recursive) contents of any client machine directory or from part of an existing version; in either case, files can be filtered out by pathname.

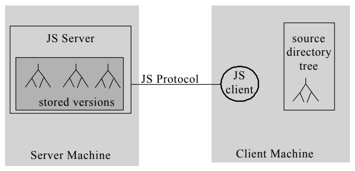
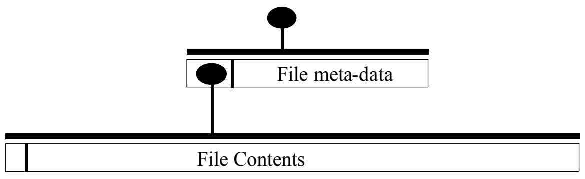


Figure 1: Incremental upload configuration

2.1 Hash-based directed acyclic graphs

An HDAG is a special kind of directed acyclic graph (DAG) whose nodes refer to other nodes by their hash rather than their location in memory. More precisely, an HDAG is a set of HDAG nodes where each HDAG node is the serialization of a data structure with two fields: the pointer field, which is a possibly empty array of hash pointers, and the data field, which is an application-defined byte array. A hash pointer is the cryptographic hash (e.g., MD5 or SHA1) of the corresponding child. Pictorially, we represent a hash pointer as a black dot that is connected to a solid bar above the node that is hashed. For example, a file can be represented using a two level HDAG:



The leaf node's data field contains the contents of the file and the root node's data field contains the file's meta-data. Using this representation, two files with the same data contents but different metadata (e.g., different names) will have different metadata nodes but share the same contents node: because nodes are referred to by hash, there can be only one node with a given list of children and data.

Continuing our example, we can extend our representation to arbitrary directory structures by representing each directory as a node whose data field contains that directory's metadata and whose children are the nodes representing the directory's members. Figure 2 below shows an example where the metadata nodes for ordinary files have been suppressed to save space; each grey box is a contents node.

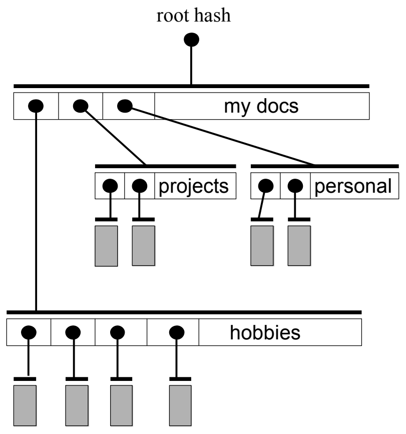


Figure 2: An HDAG representation of a directory tree

HDAGs are a generalization of Merkle trees [20]. They are in general not trees, but rather DAGs since one child can have multiple parents. Also unlike Merkle trees, their non-leaf nodes can contain data. Notice that even though a directory structure (modulo links) is a tree, its HDAG representations are often DAGs, since there are often files whose contents are duplicated in whole or in part (see chunking in Section 2.3). The duplicated files or chunks will result in two or more HDAG nodes pointing to the same shared node.

2.2 Properties of HDAGs

We say that an HDAG is *rooted* if and only if there is one node in that HDAG that is the ancestor of all the other nodes in the HDAG; we call such a node the HDAG's *root node* and its hash in turn the HDAG's *root hash*. An HDAG is *complete* if and only if every one of its nodes' children also belongs to that HDAG; that is, there are no 'dangling' pointers. Figure 2 above is an example of a rooted, complete HDAG.

HDAGs have a number of useful properties.

Automatically acyclic: Since creating an HDAG with a cycle in the parent-child relation amounts to solving equations of the form

$$H(H(x;d_2);d_1) = x$$

where H is the underlying cryptographic hash function, which we conjecture to be cryptographically hard, we think it is safe to assume that any set of HDAG nodes is cycle free. All of the HDAGs we generate are acyclic barring a hash collision and it seems extremely unlikely that a random error would corrupt one of our HDAG nodes, resulting in a cycle.

Unique root hash: given two rooted, complete (acyclic) HDAG's H_1 and H_2, they are the same if and only if their root hashes are the same. This is a generalization of the 'comparison by hash' technique with the same theoretical limitations [16]; in particular, this property relies on the assumption that finding collisions of the cryptographic hash function is effectively impossible. More precisely, it stems from the fact that a root hash is effectively a hash of the entire HDAG because it covers its direct children's hashes which in turn cover their children's hashes and so on. By induction, it is easy to prove that if H_1 and H_2 differ yet have the same root hash, there must exist at least two different nodes with the same hash.

Automatic self assembly: Because all the pointers in an HDAG are hashes, given an unordered set of HDAG nodes we can recreate the parent-child relationship between the nodes without any extra information. To do this, we first de-serialize the nodes to get access to the hash pointers. We then compute the hash of every node. Now we can match children with parents based on the equality of the hash pointer in the parent with the hash of the child.

Automatic structure sharing: Not just single nodes are automatically shared within and between HDAGs; sub- DAGs representing shared structure are as well. Consider Figure 3 below; it shows two snapshots of the same directory tree taken on adjacent days. Only one file (labeled old/new file) changed between the snapshots. Every node is shared between the two snapshot representations except the modified file's content node, its metadata node (not shown), and the nodes representing its ancestor directories. In general, changing one node of an HDAG changes all of that node's ancestor nodes because changing it changes its hash, which changes one of the hash pointers of its parent, which changes its parent's hash, which changes one of the hash pointers of its grandparent, and so on.

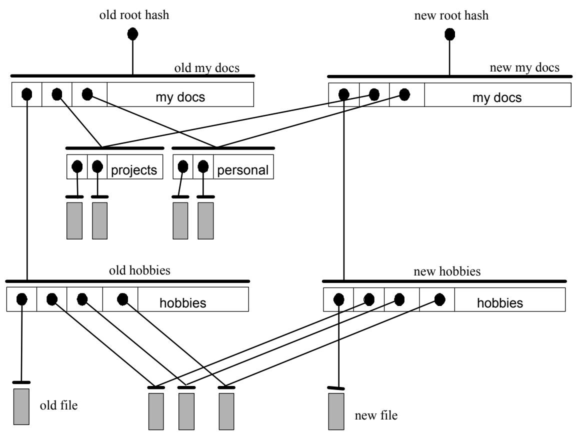


Figure 3: Structure sharing between HDAGs

2.3 Snapshot representation

The snapshot representation described in Section 2.1 has the major drawback that if even one byte of a file is changed, the resulting file's content node will be different and will need to be uploaded in its entirety. To avoid this problem, we break up files into, on average, 4 KB pieces via content-based chunking.

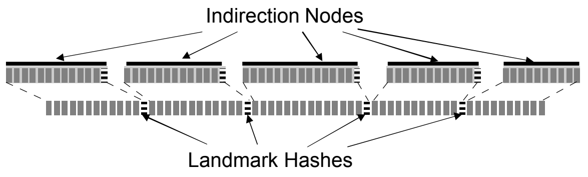
Content-based chunking breaks a file into a sequence of chunks based on local landmarks in the file so a local modification to the file does not change the relative position of chunk boundaries outside the modification point [21,22]. This is basically equivalent to breaking a text file into chunks at newlines but more general; editing one line leaves the others unchanged. If we used fixed size blocks instead of chunking, inserting or deleting in the middle of a file would shift all the block boundaries after the modification point, resulting in half of the file's nodes being changed instead of only one or two.

We use the two-threshold, two-divisor (TTTD) chunking algorithm [13], which is an improved variant we have developed of the standard sliding window algorithm. It produces chunks whose size has smaller variance; this is important because the expected size of the node changed by a randomly-located local change is proportional to the average chunk size plus the variance divided by the average chunk size. (Larger chunks are more likely to be affected.)

2.3.1 The chunk list

With chunking, we also need to represent the list of hashes of the chunks that make up a file. We could do this by having the file metadata node have the file's chunks as its children. However, the resulting metadata node can become quite large: since we currently use 17-byte long hashes (MD5 plus a one byte hash type), a 10 MB file with average chunk size of 4 KB has approximately 2,500 chunks so the list of chunk hashes alone would be 42 KB. Since the smallest shared unit can be one node, to maximize sharing it is essential to have a small average node size. With this representation, changing one byte of this file would require sending over 46 KB of data (1 chunk node and the metadata node).

We introduce the idea of chunking the chunk hash list itself to reduce the amount of chunk list data that needs to be uploaded when a large file is changed. We chunk a list of hashes similarly to file contents but always place the boundaries between hashes and determine landmarks by looking for hashes whose value = -1 mod *k* for a chosen value of *k*. We package up the resulting chunk hash list chunks as indirection nodes where each indirection node contains no data but has the corresponding chunk's hashes as its children:



We choose our chunk list chunking parameters so that indirection nodes will also be 4 KB on average in size; this corresponds to about 241 children. We use chunking rather than just dividing the list every *n* hashes so that inserting or deleting hashes does not shift the boundaries downstream from the change point. Thus, even if ten chunks are removed from the beginning of the file, the indirection nodes corresponding to the middle and end of the file are not affected.

This process replaces the original chunk list with a much smaller list of the hashes of the indirection nodes. The resulting list may still be too large so we repeat the process of adding a layer of indirection nodes until the resulting chunk list is smaller than a desired threshold, currently 2. Files containing no or only one chunk of data will have no indirection nodes. The final chunk list is used as the list of children for the file metadata node.

The result of this process is an HDAG at whose leaves are the chunks, and whose non-leaf nodes are the indirection nodes. This HDAG, in turn, is pointed to by the file metadata node. Thus, we use the chunking scheme and the indirect nodes as a natural extension of the HDAG representation of directory structures (see Figure *2*).

Under this representation, a 10 MB file has approximately 2,500 data chunks, 11 first level indirection blocks, one second level indirection block, and one file metadata node. The overhead of making a small change in this file ignoring the metadata node's contents and ancestors is the size of one chunk (~4 KB) plus the size of one first level indirect node (~4 KB) plus the size of the second indirect block (~17 bytes), which sums up to roughly 8 KB, which is much better than the 46 KB a flat representation would have required.

2.4 Efficient incremental upload and storage

Efficient incremental upload for snapshots can be described as follows: there is a site, the source, where an up-to-date copy of a directory structure exists, and another site, the target, where one or more older snapshots of the same directory structure exist. The connection between the two sites may be slow and unreliable. It is required to create a snapshot of the current contents of the source directory on the target, minimizing the transfer time and maximizing the reliability.

The properties of HDAGs make them ideal for use in implementing efficient and reliable transfers such as incremental upload: First, the automatic self-assembly property means that the HDAG nodes can be gathered from multiple sources (e.g., possibly stale caches), in any order. No matter where the nodes come from, there is only one way to put them together to get an HDAG.

Second, the unique root hash property lets us check when a transfer has successfully and correctly completed: if the target has a complete, rooted acyclic HDAG whose root hash is the same as that of the source HDAG then we have a strong guarantee that the HDAG at the target is identical to the HDAG at the source. Any extra received nodes not part of this HDAG (e.g., nodes corrupted in transit) may be discarded. If the HDAG is incomplete, it is easy to determine the hashes of missing nodes.

Third, the automatic structure sharing property ensures that many nodes will be shared between the source and target. Such nodes need not be transmitted if they can be determined to already be present on the target. This can be done by querying the existence of each source node at the target by hash (this uses much less space than sending the node itself). Fourth, by taking advantage of the unique root property, it is possible to query the existence of an entire sub-DAG with root hash *h* by sending a single hash, *h*: if the target replies that it has a complete sub-HDAG with root hash *h*, then it must have the same sub-DAG the source has; this 'compare by root' technique can be much more efficient than querying about individual nodes when a lot of structure is shared.

Combining these ideas, we get the following algorithm: The incremental upload algorithm runs on the client system. Let H be the complete HDAG representing the source directory structure. The client agent traverses H in some order and for each node N encountered, queries the remote server whether it has the complete DAG whose root is N. N is transmitted only if the answer is negative. If the answer is positive, then the children of N need not be traversed. The remote server replies with the hashes of the nodes it receives, allowing retransmission if needed. Once the client has finished traversing H, it tells the remote server to finalize the version using the HDAG with root hash H's root hash.

There is a great deal of flexibility in the order in which the nodes of H are generated and traversed. In particular, we do not require that the whole of H be in memory at the same time. Moreover, there can be multiple threads working on different parts of H concurrently. Currently, to bound the client's RAM usage even though files may be arbitrarily large, we use compare by root only for DAGs representing files whose root hashes we already know from a previous upload; for all other nodes, we query existence individually. By default, we maintain a small cache on the client of previously uploaded normal files mapping their pathname plus modification time when last uploaded to the root hash of the DAG that represented them. If these files have the same modification time as the last time they were uploaded, we can avoid regenerating their representative DAGs if compare by root succeeds.

2.4.1 Efficiency and reliability

Our current algorithm is efficient: each untouched file requires one query for compare by root, each other node (e.g., directories and the nodes in changed files) requires one query for compare by hash, and each new node additionally must be transmitted. Because queries contain only a single hash, which is 240 times smaller than the average 4 KB node size we use, we effectively send only the parts of the HDAG that have been modified since the previous snapshot. By careful design of our snapshot representation (see Section 2.3), we have ensured that small local changes to the source directory structure change as few HDAG nodes as possible.

To minimize the latency of the query-response, queries and nodes are sent in one thread while the responses are processed asynchronously in another thread; we also batch messages to reduce overhead.

HDAG nodes are computed in parallel with querying/sending. Computing HDAGs is relatively fast: a 3.2 GHz Xeon Windows PC can scan, compute HDAG nodes, and count how many unique HDAG nodes there are in an in-cache (i.e., no disk I/O) filesystem tree that contains 64 directories, 423 files, and 220 MB of data at over 18 MB/s. Accordingly, in our experience HDAG computation time is normally dominated by transmit time (slow links) or client disk scan/read time (fast links).

Because the JS stores each node only once, these same properties allow us to store multiple successive snapshots of the same directory tree in very little space; in effect, storing another snapshot requires as much space as would be required to incrementally upload that snapshot.

What happens if something goes wrong during the upload process? If some nodes get corrupted in transit, then we will detect that by comparing the returned hashes, and the nodes will be re-sent. What if the upload process is interrupted for some reason? Let us say that 70% of the way through the transfer the client crashes. All we have to do is to start the upload process again from the beginning (no client state need be kept). Since we still have all the HDAG nodes that have already been transferred on the server, very quickly the client will reach the same point in the process where the previous transfer was interrupted, and continue from there. The only time lost is the time to scan the source directory and construct the HDAG again, which is a fraction of the transfer time. Because of the strength of the cryptographic hash we use and unique root hash property, we can be very sure if the transfer succeeds that no errors have been made.

2.5 Implementation

The JS server, about 13,000 lines of C++, runs on a single Windows or Linux machine and supports multiple concurrent client TCP connections. The basic JS client is a command-line program, about 15,000 lines of pure Java, which can run on any operating system that supports Java 1.4.

The Jumbo Store, unlike other content-addressable stores [5,24,29], is an *HDAG-aware store*. That is, in addition to operations to store and retrieve the basic unit of storage (the node for JS) by hash, the JS server supports operations on entire HDAGs. For example, it supports 'compare by root' queries ("do you have a complete HDAG with root hash *h*?"), "how big is the HDAG with root hash *h*?", and the deletion of entire HDAGs (really versions). The JS server does not interpret nodes' data fields and knows nothing of snapshots. The protocol the JS speaks has no connection-specific state and all messages are idempotent, allowing easy retransmission in case of lost messages or connections.

JS data is stored in a series of large data files on disk; an in-memory hash table indexes the nodes stored in the data files by their MD5 hash. A separate file for each version contains only that version's root hashes—partial versions may have multiple roots. To support deletion, the index also maintains a reference count for each node where each version root is considered a root for the purposes of reference counting. Occasionally a background process compacts data files by copying only the nodes with a nonzero reference count to a new file. This simple reference counting garbage collection scheme works well because HDAGs are acyclic.

Due to space limitations, we will not discuss downloading snapshots or the other operations the JS client supports further except to note that we use a sophisticated tree pre-fetching algorithm to avoid pipeline stalls during downloading.

3 The Utility Rendering Service

The Utility Rendering Service (URS) is a batch utility service that performs the calculations required to render a 3D animated movie. It gives animators access to a large pool of resources to perform the rendering, and allows them to purchase rendering resources when needed. Animation is an interesting domain in which to test technologies for Utility Services because of the natural cycles in demand for resources inherent in a typical movie production cycle.

The URS does not fundamentally change the way in which an animator works; they still use the tools they are familiar with. However, it does offer the potential for a more efficient and interactive style of work because animators have access to a more powerful set of resources than they could otherwise economically afford, allowing the visual quality settings to be turned up, and allowing the animator to be more experimental because the turnaround time for scenes is reduced.

The Utility Services model is particularly attractive for small animation organizations, because it allows them to acquire computing resources at short notice when needed, allowing individuals and small teams to dynamically form and take on projects that would otherwise not be possible if only in-house computing resources were used. Because of space limitations, we will concentrate here on only the aspects of the URS that are relevant to the use of the JS.

3.1 User model

Animators use a commercial content creation application called Maya® [3] to create the digital models that define their 3D animated movie, including the shape and movement of characters, backgrounds, and objects, and associated textures, lighting, and camera definitions. Maya uses over a dozen file formats including a variety of image formats (e.g., JPG and TIFF) and several proprietary formats; most of these are binary formats, although a few are ASCII (e.g., the MEL scripting language).

To interact with the URS, animators use a Java application called the URS Client, running on one or more of their computers. The URS Client allows users to upload input data, submit rendering jobs, monitor the progress of jobs, download rendered frames, and manage the data stored on the server.

We imagine a dynamic, competitive market for Utility Services, where customers may only subscribe to a service on demand and for limited periods, based on factors such as price and functionality. Accordingly, the barrier for successful subscription to, and use of, a service needs to be low. Towards this end, the URS client is written in pure Java for operating system portability, automatically works through firewalls, is easy to download, and is self updating.

The URS separates the tasks of uploading animation models, rendering models into frames, and downloading frames for viewing, allowing them to be performed independently and, in many cases, in parallel. Uploading input data (a directory tree specified by the user containing a consistent set of files that can be rendered) results in a new snapshot of the input data stored at a URS server; these snapshots are referred to as "versions" by the URS system. Versions remain until explicitly deleted by a user but are subject to an overall space quota. Note that the root of the input data directory tree can be changed each time a new version is created, so, unlike a source-code versioning system like CVS, the structure of the files and directories may change radically from one version to the next.

To render frames, an animator submits a new job request against a specific version, specifying the name of a scene file within that version and the frame numbers to compute. A job can be submitted against a version any time after its uploading has been initiated. Allowing multiple jobs per version and rendering multiple versions at the same time greatly increases flexibility. For example, an animator may wish to interactively make several changes to a character model and experiment with which looks best, and have the rendering service compute each possibility simultaneously.

Newly rendered frames are downloaded in the background by default as they become available. Alternatively, animators may explicitly request when and which frames should be downloaded.

3.2 Architecture

The overall architecture and data flow of a URS instance is shown in Figure 4 below. A server-side subsystem of URS, the Asset Store, manages the transfer and storage of the input and output data. The Asset Store consists of two processes (Asset Manager and Jumbo Store) and three internal storage areas, each with an associated storage quota that users must keep within.

The Version File Store stores the data managed by the JS server process; it contains in compressed form the available URS versions and possibly a partial version in the process of being uploaded. The Output Content Store stores the rendered frames generated by processing nodes.

The remaining storage area is the Version Cache (VC), which stores a subset of the versions held in the Jumbo Store in their fully expanded form, ready for use by the processing nodes. The VC is needed because the JS currently only supports uncompressing an entire snapshot at a time, a time-consuming operation, and there is not enough room to keep every version in expanded form.

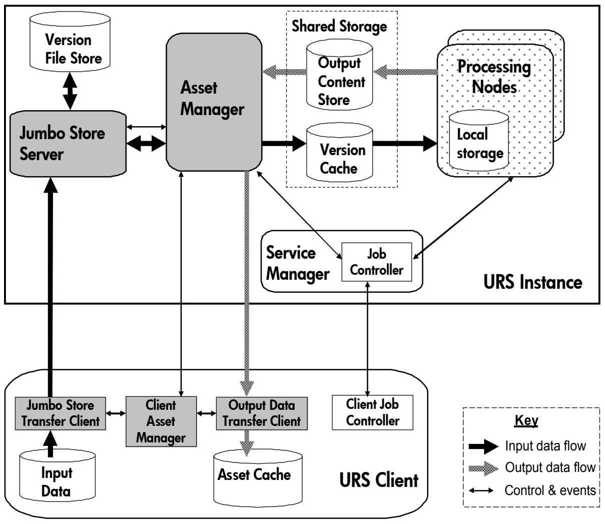


Figure 4: URS architecture and data flow

The lifecycle and state of input data versions is managed by the Asset Manager. Versions have a well-defined lifecycle, representing the stages of creation, transfer, archival to Jumbo Store, restore to VC, deletion from VC, and removal. Important changes to the Asset Manager state are held persistently in a database so that state can be fully recovered on service instance restart even after failure. Incomplete asynchronous operations on input data versions, such as upload, extraction, or deletion, are either cancelled or completed as appropriate. To keep the design simple, only one upload is permitted at a time per service instance.

3.3 Client-server communication

Communication between all components running in the URS Client and those in the URS is implemented over a single Secure Socket Layer (SSL) encrypted socket connection made from the client. This gives automatic client firewall traversal, the ability to easily terminate in a single operation on the server all interactions with a specific user, and, similarly, the ability to reestablish communication in the event of temporary connection failure with a single operation. However, the disadvantage is that all data, control, and event protocols must be multiplexed down a single channel.

All client-server communication, for data, control, and events, is implemented over a simple object passing and addressing abstraction called the Message Object Broker (MOB), which is layered above the SSL socket. The MOB allows serialized Java objects to be exchanged across the socket to named recipients on the remote side, and offers a variety of call semantics such as request-reply, buffered writes, and direct object passing. It also implements a simple keep-alive mechanism, shared by all protocols using the MOB to detect connection failures. The pure Java implementation strategy, and the use of serialized Java objects, did not prove to be a problem for acceptable performance of bulk data transport.

4 SE3D Results

4.1 Setting

The URS was offered to 11 small teams of animators during an animation showcase, called SE3D, to create high-quality short animations. The SE3D animation showcase was a unique experiment, conducted over a period of 10 months, giving new, creative talent from the animation industry access to a set of research technologies for Utility Services, together with a large pool of computer resources. The trial involved up to 120 dual 3 GHz Xeon processor servers, each with 4 GB RAM, and a total of 4 TB of storage. The URS server-side components, including the Jumbo Store servers, were deployed in a data centre in the US, while the animators were all located in the UK. Thus all data transfers had to traverse the public Internet over a transatlantic link.

There was considerable variation between teams in working methods, kinds of Internet connections, number of animators using the URS, how often and how many times they uploaded, how many client machines they used, how big their movie source was, and the like. Table 1 below summarizes each team's use of the URS upload facility; to preserve privacy we have assigned teams service instance numbers in order of increasing movie source size. Here, ‘uploads’ is the total number of uploads attempted by that instance and ‘logged’ is the number of those uploads for which we have correctly logged information—because JS was added to the URS after SE3D started and because some early bugs caused bad logging, we do not have useful information for some early transfers; in particular, we have no trustworthy data for instance 0 so it is omitted from the rest of this paper. The remaining two columns give the average version size (i.e., movie source size) and average number of files involved in the correctly logged uploads for that service. Note that size here refers to the size of the version on the client, not the amount actually transferred to or stored at the Jumbo Store.

service instance	uploads	logged	average size (MB)	average # files
0	43	0		
1	124	17	92.0	24.7
2	87	68	109.8	21.2
3	287	286	143.7	5025.5
4	217	122	342.7	145.0
5	379	263	351.4	76.4
6	32	29	352.3	91.4
7	229	209	360.1	99.8
8	55	42	1873.6	225.6
9	125	109	2498.5	773.6
10	202	169	3046.3	4709.3
avg	161.8	119.5	917.0	1119.3
all	1780	1314	859.3	1929.0

Table 1: Use of the upload facility

All but one service exploited the Jumbo Store's ability to hold multiple versions in order to render multiple versions at the same time. Although most services rendered a maximum of three or four versions simultaneously, two services rendered 7 and 10 respectively versions at the same time.

4.2 Reliability and robustness

Transferring gigabytes of data via TCP without higher level end-to-end checking and retransmission is problematic: given TCP's 16-bit checksum and assuming a 1% packet error rate and 1500 byte packets, we expect an undetected data corruption error to occur once every 9.2 GB of data. Indeed, the authors were unable to check out a 12 GB Subversion repository over the transatlantic cable due to repeated network errors and Subversion's inability to restart incomplete transfers where they left off. By contrast, our first attempt to copy the same data via JS worked perfectly.

When used independently, Jumbo Stores verify each received chunk using cryptographic checksums, requesting retransmission as needed, to handle transmission errors. They also reconnect transparently should a TCP connection be broken due to an error or a timeout. Accordingly, neither kind of error requires restarting an upload.

As incorporated into the URS, Jumbo Store traffic is sent over SSL using a supplied MOB connection. Because of SSL's cryptographic checksums, any data corruption results in a broken connection. Unfortunately, while the URS can automatically reestablish a new connection, it cannot do so in a manner transparent to the MOB's clients, which include the JS. It can, however, automatically restart at the beginning an upload aborted due to a broken connection. Because the JS upload protocol does not resend data already on the Jumbo Store, we quickly scan forward to the furthest point the upload previously reached.

During SE3D, there were 262 restarts, the vast majority of which (251) were for service instance 5, whose Internet connection appears to have been unreliable at times—1 transfer restarted 58 times before the user stopped it. Inspecting the logs shows that 91.7% of the uploads succeeded, 7.8% of the uploads were aborted by users before they completed, and 0.4% of the uploads failed due to URS problems unrelated to the JS. At most 12% of the user aborts can be attributed to frequent restarts. The remaining aborts are presumably due to users realizing they had made a mistake or wishing to upload instead an even newer version. If we count the later as successes, then the overall URS upload success rate exceeds 98.6%.

The only version data loss we suffered occurred early on due to a bug in the JS server's garbage collector. The bug was quickly fixed and we were able to recover much of the data from the URS Version Cache.

4.3 Compression

For the purposes of this and Section 4.4, we analyze only the 1092 uploads (83% of the correctly logged JS uploads) that succeeded, did not restart, and immediately follow a successful upload. This is necessary to ensure meaningful statistics; e.g., an aborted upload may have partially uploaded a snapshot, making the next upload seem artificially efficient.

Table 2 below shows average compression ratios (i.e., compressed size/uncompressed size) of various kinds for each of the instances (1-10), all the instances treated as a single service (all), and the average service instance average compression ratio (avg, the average of the individual instance numbers). The all numbers differ from the avg numbers because they more heavily weigh instances with large numbers of uploads/versions. We will quote both numbers as avg # (all #).

service instance	upload	within version	across versions	both
1	20%	39%	37%	16%
2	9.3%	48%	16%	10%
3	6.7%	69%	9.7%	6.8%
4	1.4%	41%	3.7%	1.7%
5	2.1%	30%	5.6%	2.1%
6	19%	81%	21%	17%
7	1.6%	34%	4.1%	2.0%
8	1.6%	75%	9.1%	7.2%
9	0.52%	28%	15%	3.8%
10	1.1%	36%	1.5%	1.0%
avg	6.3%	48%	12.3%	6.8%
all	3.5%	44%	7.3%	4.0%

Table 2: Various compression ratios

The upload column shows the average upload ratio of the actual number of data and metadata bytes uploaded over the total number of data bytes in the snapshot being uploaded. Thus, a conservative approximation of our upload compression ratio is 6.3% (3.5%); equivalently, our upload compression factor (1/ratio) is 16x (29x). While analyzing the logs, we discovered a performance bug: a second write of a block while its first write was still in progress could result in that block being transmitted twice. We conservatively estimate that had this bug been fixed beforehand, our upload compression would have instead been 5.5% (3.2%) or 18x (31x).

The 'within version' column shows the average version storage compression ratio under the restriction that no sharing is permitted between versions; the restriction is equivalent to requiring each version to be stored on a separate Jumbo Store by itself. These numbers—48% (44%) or 2.1x (2.3x)—are surprisingly good and indicate that movie sources are fairly redundant.

The 'across versions' column attempts to measure the degree of storage compression due to sharing between versions (of the same service instance) rather than within versions. It shows the average ratio of the additional storage required to store a new version on a Jumbo Store containing all surviving previous versions over the amount of storage required to store that version separately. Between version compression gives us 12.3% (7.3%) or 8.1x (14x). Note that the degree of storage compression possible due to sharing between versions depends on user deletion behavior: if users delete all versions before each upload, for example, we will get no storage compression due to sharing between versions.

The 'both' column shows the average actual version storage compression ratio we achieved, including the savings from sharing within versions and across versions (of the same service instance). We achieved a storage compression ratio of 6.8% (4.0%) or 15x (25x). These numbers mean that 10 successive versions (one full and nine incrementals) can be stored by a JS in the space required to store one uncompressed version.

The astute reader will have noticed that our storage compression ratio is slightly worse than our upload compression ratio; this is because our URS upload code keeps a copy of the last (partial) upload in a staging area on the server; this reduces the amount of data that must be transferred, but does not count as previously stored data for the purpose of determining how much new data has been added to the store.

4.4 Speed

The median time from an animator requesting a version be uploaded to all of that version's bits being known to be present on the Jumbo Store (upload) is shown for each service instance in Figure 5 below; the average median time to upload a version (avg) was 4.4 minutes and the median time for all uploads (all) was 1.8 minutes.

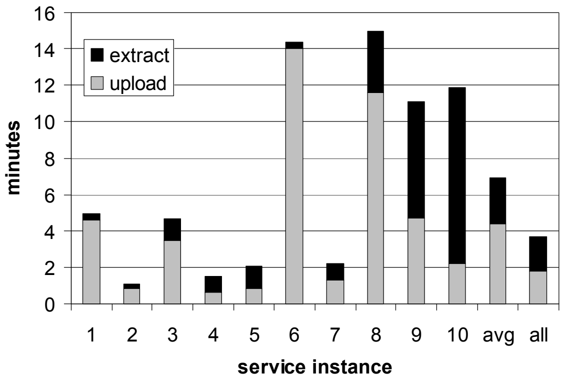


Figure 5: Median upload and extraction times

Also shown is the median time from the request until the new version is available in the URS's Version Cache (upload+extract), which is required before rendering can start. Extracting a version involves downloading that version from the Jumbo Store to the Version Cache located on the same machine. Because the Version Cache copy is uncompressed, extraction time is necessarily proportional to the uncompressed size of the version rather than the much smaller amount of data actually sent/stored in the Jumbo Store. Requiring extraction is suboptimal; in the future we may be able to eliminate it and the Version Cache altogether in favor of rendering directly from the Jumbo Store data via a filesystem abstraction. Extraction took 2.5 (2.0) minutes, yielding an overall transfer time of 6.9 (3.8) minutes.

To put this in perspective, downloading a single frame (~900 KB) took 10 seconds on average. Although an average of 250 frames were downloaded per version (~50 minutes of total download time), most of these would have been downloaded either in the background while working or overnight—a small sampling of frames usually suffices to find errors/verify changes. Rendering a frame took a few minutes to several hours depending on the complexity of the frame (e.g., fur slows things down). Frames can be rendered in parallel, however.

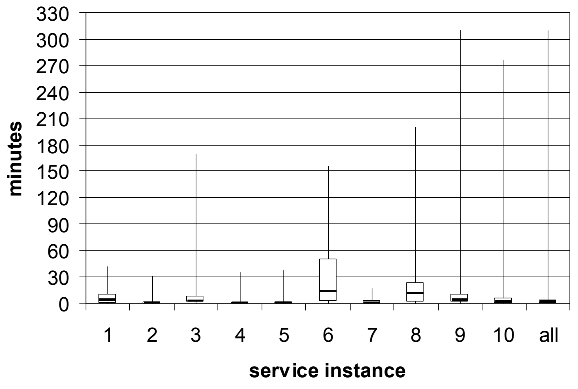


Figure 6: Distribution of upload

We quote median rather than mean values in this subsection because the underlying distributions are highly skewed toward smaller values; Figures 6 and 7 provide information about the distribution of upload for each service instance using box plots. Each box ranges from the 25th percentile value to the 75th percentile value and is divided into two parts by a line at the median (50th percentile) for value. Lines extend vertically from each box to the minimum and maximal values of the given distribution. The high tails of the upload distributions drop off roughly inversely to time.

Upload times are affected by the actual amount of bandwidth available and the amount of data that needs to be uploaded. Actual bandwidth, which we were unable to measure, depends on the speed of the animator's connection and the amount of congestion experienced from other programs on the same computer, neighbors in the case of shared connections (e.g., cable modems), and other users of the transatlantic cable. Except for two of the instances, most of the variance in upload times for an instance is due to variance in the amount of data that needed to be uploaded; Figure 8 shows the distribution of the sent user data size for each instance. The average median amount was 3.8 MB and the median amount for all uploads was 0.80 MB.

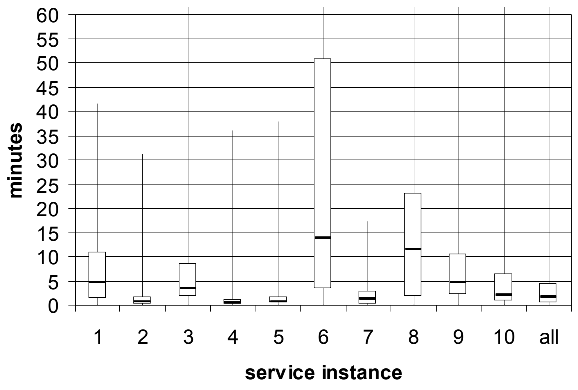


Figure 7: Detail of bottom of Figure 6

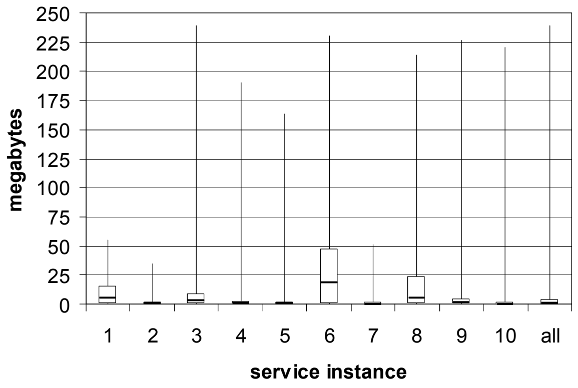


Figure 8: Distribution of amount of user data sent

Although we do not know what the actual raw maximum bandwidth available for any given upload was, we can estimate the effective bandwidth (total size of user data sent/time required) for each instance; Table 3 below shows the results of applying linear regression to each instance's sent user data size, upload time pairs excluding a few outlier points whose residual's were more than three standard deviations from the norm. For example, we predict service 1 sending 5 MB of file data would take 25 + 5*1024*8/187 = 244 seconds. Fit was good (high R^2) for all service instances except 5 and 7; recall that service 5 had numerous connection problems.

These bandwidth calculations do not include control messages, queries, metadata, or TCP overhead. Overhead includes both setup/finishing steps and work proportional to the size of the version being uploaded rather than the amount of data being transferred (e.g., queries).

service instance	bandwidth (Kbits/s)	overhead (s)	R^2
1	187	25	0.998
2	155	18	0.989
3	198	81	0.989
4	706	29	0.981
5	638	59	**0.307**
6	200	166	0.983
7	184	66	**0.601**
8	165	103	0.954
9	101	129	0.999
10	192	176	0.929

Table 3: Estimated effective bandwidth for each service instance

4.5 User feedback

Extensive interviews were conducted with the teams of animators before and after SE3D. We report here mostly the parts relevant to the use of the Jumbo Store in the URS. The interview subjects agreed unanimously that the URS was easy to setup and install; 33% thought it met expectations while 56% thought it was simpler and easier than expected. More telling, almost all subjects said they would be interested in it for commercial use. The faster rendering speed and the ability to be operated remotely of the URS led several of the animators to change their working practices; one animator was in The Hague for nearly 6 weeks and continued working by using his laptop in Internet cafés.

Animators are not technical people. They are very visual/tangible thinkers; this led to some difficulties with the programmer-influenced user model and interfaces. We discovered after SE3D was over that there was a fair amount of confusion on how uploads worked and what versions were. Some animators mistakenly thought upload time was proportional to the amount of data in their upload directory; this caused some of those to take care to "upload" only the fraction of the movie source relevant to a given rendering step by copying the relevant files from their actual source directory.

There was also confusion about the meaning of "version". In the mind of the animators, a version is a snapshot of a set of files defining a project that have reached some key milestone in the project. They were thus puzzled when a minor change produced a new version. The animators' normal work practice was to keep each revision of a given scene file by using related filenames (e.g., clouds.1, clouds.2, etc.); some insisted on this practice even though they thought (erroneously) that it was hurting their upload performance.

5 Comparison with Rsync

The best alternative to the Jumbo Store we know of for uploading files across a low bandwidth connection is Rsync [26], an open source utility that provides fast incremental file transfer. Accordingly, we compared uploading a subset of the SE3D data across the transatlantic cable using the Jumbo Store (independently, with no SSL) and using Rsync.

The data used was a subset of the versions uploaded by the animators; more precisely, the data is from a copy made during a maintenance window late in SE3D's life of the Jumbo Stores' data files. It is thus lacking any versions uploaded after or deleted before that point. Although this is the most representative data we have, it is likely less compressible than the actual sequence of versions uploaded during SE3D because it is missing intermediate versions. The data used contains 441 versions distributed as follows:

service instance	versions		service instance	versions
1	90		6	15
2	41		7	84
3	12		8	**6**
4	76		9	**2**
5	99		10	16

Note that we have very few versions for service instances 8 and 9.

The uploading was done from a 1.8 GHz Pentium 4 PC with 1 GB of RAM running Suse Linux 9.1 in Palo Alto, California to an 800 MHz Pentium III PC with 512 MB of RAM running Red Hat 9 Linux in Bristol, England. Both PCs are inside the HP corporate firewall, but the connection between them runs through the public Internet and over the transatlantic cable. Previous experiments indicate that the transatlantic cable is the bottleneck for this connection, with a peak bandwidth of slightly less than 2.7 Megabits per second (2800 Kb/s).

Our experimental procedure was as follows: for each service instance, we first emptied the destination directory (for Rsync) or store (for JS). We then uploaded each version belonging to that instance in turn in the order they were originally uploaded. Every upload for a given service instance other than its first thus had the potential to be an "incremental" upload. We used tcpdump and tcptrace to record the elapsed wall time and number of unique bytes sent (i.e., the total bytes of data sent excluding retransmitted bytes and any bytes sent doing window probing) and received of each upload. Due to time constraints (a full run through of all the data for a single method takes weeks), we were only able to repeat this procedure once per upload method.

Figure 9 below compares the number of unique bytes transmitted (i.e., sent or received) by Rsync, by our original Jumbo Store with the block retransmission bug fixed (JS), and by an improved version of the Jumbo Store (JS+), which we describe shortly. For ease of comparison, we present normalized numbers where Rsync's performance is designated as 1.0. In addition to per service instance numbers, we also show numbers for combining all the uploads (all, with emptying when switching instances) and the median of the instance numbers (med). Overall, JS transmitted 52% (med 53%) or 1/1.92 (med 1/1.89) of the bytes that Rsync did.

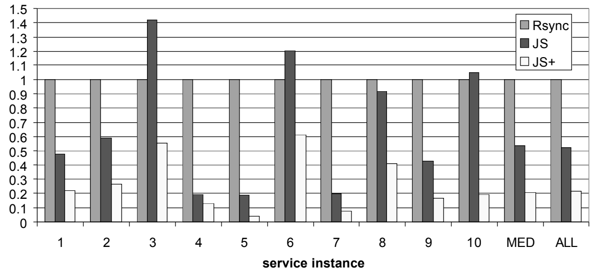


Figure 9: Total bytes transmitted for each method

We invoked Rsync with the "-compress" option, which is recommended for low bandwidth connections and has the effect of gzipping data before it is transmitted. This compression is on top of Rsync's delta compression, which attempts to send only the portions of files that differ. Our experiments indicate that failing to use -compress results in Rsync sending 190% more bytes overall (all) on this data set.

Inspired by this result, we created an improved version of Jumbo Store (JS+) that gzip's each set of chunks to be sent during transmission; by default, each set of sent chunks has 50 ~4 KB chunks for a total size of ~200 KB uncompressed. This change substantially increased performance: JS+ transmits 1/2.5 (med 1/2.6) the bytes that JS does and only 21% (med 21%) or 1/4.7 (med 1/4.8) of the bytes that Rsync did.

If we consider only the "full" uploads, JS+ transmits only 39% (med 55%) of the bytes that Rsync does. Considering only the "incremental" uploads instead, JS+ transmits only 15% (med 12%) or 1/6.7 (med 1/8.3) of the bytes that Rsync does.

Figure 10 below compares JS+'s performance to Rsync's using both bytes transmitted and time elapsed;

as with Figure 9, we have normalized so that Rsync's performance is 1.0. Measuring by time, JS+ is only 3.4x (med 3.1x) faster than Rsync. We estimate using linear regression that overall (all) actual bandwidth (unique bytes transmitted/elapsed time) was 2.44 Mb/s for JS+ and 2.49 Mb/s for Rsync with overheads of 6.8 seconds for JS+ and 3.0 seconds for Rsync.

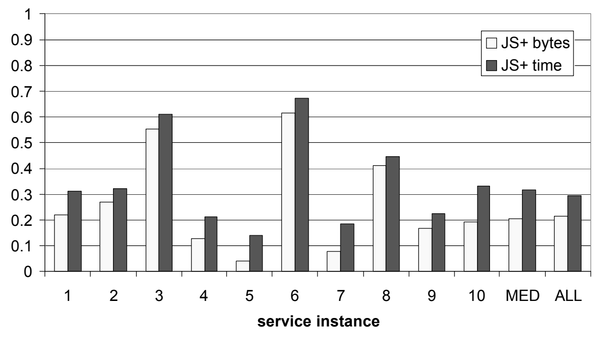


Figure 10: Normalized JS+ performance vs. Rsync

6 Discussion of results

The level of compression and reliability achieved by a system is heavily dependent on the actual data to be compressed and the setting it is deployed in: it is easy to achieve 100% reliability in a controlled lab setting or good compression by using synthetic data created from the same distribution your compression algorithm was designed to compress. SE3D represents the gold standard in test data: large amounts of real data collected over a long time from real users using the system for its intended purpose. Because the services were isolated from each other for security and performance reasons, SE3D can be viewed as a series of 10 natural experiments. The large variance in outcomes between experiments—the upload compression ratio varied by a factor of 38 and the storage compression ratio by a factor of 16, for example—indicates that animators differ greatly in the characteristics that affect our system and Rsync's performance. We expect our system to work as well or better for longer movies (the SE3D animators created ~5 minute shorts) because movies are built from short scenes and because there is more opportunity for reuse of characters, sets, and the like. The performance of Jumbo Store on other domains is currently unclear; we are conducting experiments to address this.

The reliability of the Jumbo Store itself once we fixed some initial bugs was perfect: all upload problems were due to the URS, either directly or indirectly (i.e., the need for restarts due to MOB limitations), or nonworking Internet connections beyond our control. Clearly, the animators could have benefited from a better explanation of how the upload process works: the error-prone process of managing separate upload and working directories used by some of them could have been avoided. Likewise, future versions of the URS should provide more workflow support and make a distinction between "major" (meaningful to animators) and "minor" (aka, JS) versions.

Aside from reliability, the most important metric for an upload system is average upload time. We estimate that our original system is 24 times faster than one that does no compression: without compression and at the observed effective bandwidths, the average service median upload would have taken 2.8 hours. The possible productivity improvements from switching from several hours per upload to several minutes should not be underestimated. Had we deployed instead our improved version of Jumbo Store (JS+), we estimate it would have speeded things up 1.5 times to 35 times faster than no compression and an average median upload time of 2.3 minutes (4.8 minutes with extraction). The variance in the amount of data that needs to be uploaded and hence the upload times is not too surprising if we consider the animation process similar to that of program development: the changes between program runs are mostly small, but occasionally the programmer makes a major change that cannot be tested incrementally.

The Jumbo Store—especially the improved version—clearly outperforms Rsync for the SE3D-derived benchmark. Primarily this is because JS+ sends only 1/5 the amount of data that Rsync does. We attribute much of this reduction to the JS's ability to exploit sharing across files with different names, both within versions and across versions. Because Rsync computes pair-wise delta's between files with the same path names, it cannot exploit this sharing. Although we did not investigate the causes of this sharing, it is clear that one cause is some animators' use of numbered file versions (e.g., "foo.1", "foo.2", etc.): because each new file version has a new name, Rsync sees no sharing.

When Rsync is used to upload data to Linux, hard links can be used to store multiple snapshots in a compressed manner [25]: if a file is unchanged from the last snapshot, Rsync can simply create a hard link to the last snapshot's copy instead of creating a new copy. This provides limited compression as even a one byte change prevents any sharing and there is no compression within files or between files with different names. The low degree of compression does mean that no extra extraction step would be needed if used with the URS.

7 Related work

Content-addressable stores (CASs) [5,10,11,15,19,24, 29] allow stored items to be retrieved by their hash. Flat CAS systems treat the items that they store as undifferentiated blobs: the interpretation of each item is entirely up to the store's clients. The Jumbo Store is a non-flat CAS system: while it does not interpret nodes' data fields, it is HDAG-aware and does interpret nodes' children pointers. This allows it to support important operations like 'compare by root' and version deletion that otherwise would require clients to perform thousands to millions of more basic operations, which is especially problematic over low bandwidth connections.

Venti [23], a versioned file store, and CFS [9], a read-only distributed file store, use HDAG-like structures at the application level but rely on a flat CAS for storing their data. SUNDR [19] and ROFS [15] use an HDAG encoding of directory structures to ensure the integrity of the contents on untrusted servers. They take advantage of the unique root hash property by signing just the root hash with the private key of a legitimate authority. Any client with access to the public key of that authority can then verify the integrity of the contents. An intruder without access to the authority's private key cannot modify the contents without being detected, since modifying the contents will change the root hash. These systems [9,15,19,23] do not use chunking or take advantage of the properties of HDAGs for facilitating directory synchronization. While SUNDR offers multiple versions, it does not seem to support the deletion of versions once a short time period has elapsed.

THEX (Tree Hash Exchange Format) [7] specifies a way to create a Merkle tree from a byte sequence, encode the resulting tree and encapsulate it in an XML file. Its main purpose is to allow verification of fragments of the byte sequence from different sources while trusting only one source to provide the root hash of the tree. It is meant to be used in conjunction with BitTorrent-like protocols to improve the detection and retransmission of corrupted blocks before the whole byte sequence is retrieved. Unlike our approach, THEX encodes the whole Merkle tree for a byte sequence in one message, so there is no sharing of intermediate nodes. As a result, compared to a flat representation of the block chunks, it actually increases the communication overhead for the file. THEX does not have any mechanism for encoding directory nodes.

Duchamp [12] describes a toolkit for synchronizing directory structures accessed as NFS mounts. A hash tree encoding of the structure of a directory tree, similar to our HDAGs, is used for facilitating the rapid synchronization of the 'master' and 'slave' directories. While Duchamp's toolkit supports the break up of large files into smaller pieces, it does not use chunking or indirect nodes for efficient file synchronization, and it does not support multiple versions. BitTorrent [4] uses fixed-sized blocks and compare by hash to transfer files.

Unlike these systems (Venti, CFS, SUNDR, ROFS, THEX, Duchamp, and BitTorrent), many recent systems including LBFS [21], CASPER [30], Pastiche [8], and TAPER [18] use chunking and compare by hash to optimize communication and/or storage requirements when multiple versions of a file exist. In the case of LBFS, this is done to speed up the transfer of files where the target may have already seen earlier versions of the files (or at least fragments of them). All of these systems use a flat sequence of hashes to represent a file and thus would benefit from the use of indirection nodes and HDAGs. They would also benefit from upgrading to our TTTD chunking algorithm.

TAPER [18] uses hash tree encodings of directory structures to facilitate directory synchronization. The hash trees used by TAPER are somewhat different from the HDAGs described in this paper. They do not encode the file and directory metadata, and as a result cannot directly be used for verifying the integrity of the directory structure on the target. The hash of intermediate directories is determined by an in-order traversal of all the children of the corresponding node, concatenating all the children's hashes as well as traversal direction information (e.g., H("up")), and taking the hash of the concatenation. This is a more computationally expensive procedure than that used by our encoding, with no apparent advantage. While TAPER uses chunking for file synchronization, it does not treat the resulting chunks as children of the file nodes in the hash tree. It uses a separate LBFS-like algorithm for file synchronization, and does not use indirect nodes to share sequences of long files. As a result, the whole hash sequence needs to be transmitted even if only one chunk has changed. TAPER does not support versioning.

Comparison with LBFS: Compared with LBFS, our combination of compare by root and indirection nodes significantly increases the bandwidth efficiency of transferring files. Where with LBFS the server has to be queried for every chunk, with our algorithm whole sub-trees of the directory structure can be skipped when an identical copy exists on the server. Moreover, because LBFS uses flat hash lists for its file representation, the whole file representation must be sent over the wire even if the modification to the file is small.

LBFS is a file-level protocol: it does not have any representation of the directory structure. As a result, directory data is neither compressed, nor verified, in its protocol. Our protocol, by contrast, which uses a HDAG-based representation of directory structure, is efficient, robust, and fault tolerant at the directory level. LBFS does not provide for the efficient storage of multiple versions of files or snapshots.

Note that distributed filesystems like LBFS are not suitable for the URS or many other synchronization applications because of their poor responsiveness (the trans-Atlantic cable has high latency), need for constant connectivity, and failure to respect the fact that the client's contents not the server's are the ground truth. Providing a disconnected mode would help but negates the primary value of using a distributed file system for synchronization: sending changes as they are made rather than all at once at the end. Supporting multiple operating systems is substantially more difficult with a distributed file system approach.

Comparison with Rsync: Even though Rsync is a directory tree synchronization protocol, it does the synchronization through pairwise file comparisons based on files' pathnames. As a result, it completely misses intra-source sharing (when multiple files in the source's directory tree share significant content) and is completely stumped when directories or files are renamed or moved. Our representation and algorithm are insensitive to such changes, and can naturally detect and exploit intra-source sharing when it exists. In terms of reliability and robustness, Rsync verifies data only at the sub-file level; it lacks any form of overall verification.

Comparison with Grid: Solutions exist in the Grid [14] community to synchronize, manage, and process data [1,2,6,14,28,31]. These approaches target a different problem: high-performance computing applications with relatively static, huge data sets (possibly terabytes), and (multi-)gigabit-class connectivity. Typical use cases in this environment do not require support for simultaneous, overlapped processing of multiple versions of frequently-updated input content.

8 Future Work

There a number of ways the Jumbo Store and URS can be improved:

Lazy extraction: Currently before a processing node can start rendering, the entire relevant version must be extracted from the Jumbo Store to the Version Cache. This can lead to significant delay as well as unnecessary work if not all of that version's files are needed for the current rendering task. A better solution would be to extract files only as needed directly from the Jumbo Store. Accordingly, we are working on a remote filesystem interface for JS so that clients (in this case the processing nodes) can directly mount read-only the filesystems contained in JS versions. It is not clear that this will entirely eliminate the cost of extraction as the lazy interface may be slower than directly accessing an uncompressed version due to poorer locality.

Trickle upload: The URS client currently sends changes only when the user explicitly requests an upload of a new version; consequently all the changes since the last upload must be transmitted before rendering can commence, leading to delays. A more responsive system would use trickle uploading where a background task periodically scans the user's data and optimistically sends any new data chunks to the Jumbo Store. When the user finally requests an upload, few chunks would likely remain to be sent, allowing rendering to start sooner. Sent chunks that were superseded by later changes would be freed later during garbage collection.

Larger multi-user stores: Our current Jumbo Store server uses an in-memory chunk index, which limits its holding capacity to tens of gigabytes (compressed) assuming ~4 KB chunks. While more than adequate for a single SE3D service, other utility computing services may have larger jobs or wish to share a single JS instance between many services. To handle this, we are developing a new JS server that uses a disk based index and has support for access control and allocating resources among users.

9 Conclusion

In this paper we described an HDAG-aware content addressable store, the Jumbo Store. An HDAG is an immutable data structure for representing hierarchical data where hash pointers are used to connect the nodes. We built an incremental upload mechanism for directory snapshots that takes advantage of the unique root hash, automatic self assembly, and automatic structure sharing properties of HDAGs and the store's HDAG support, to efficiently and reliably upload large directory snapshots over slow and unreliable public internet connections. The store has built in facilities for the creation, retrieval and deletion of versions, which are named HDAGs. We used these facilities to build a system for efficiently storing many versions of a directory tree.

The ability to transmit large quantities of data over the slow Internet connections typical of many organizations, to be processed by Utility Services, is often perceived as a barrier for widespread adoption of the utility model. The JS was successfully used within a Utility Rendering Service, used to create 3D animated movies, and demonstrated that interactive, data-intensive services can work well even over low-bandwidth connections. The speed of upload offered by the storage system encouraged users of the service to work in an experimental fashion to try new ideas containing variations of data content. The synchronization and storage performance of the JS with the real-world data produced by small teams of animators has been analyzed and compares favorably with other competing approaches, both in the URS environment and under controlled experimental conditions.

References

[1] W. Allcock et al. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In *Proceedings of 2001 IEEE Mass Storage Conference, 2001.*

[2] W. Allcock, J. Bester, J. Bresnahan, S. Meder, P. Plaszczak, and S. Tuecke. GridFTP: Protocol extensions to FTP for the grid. GWD-R (Recommendation), April 2002. Revised: Apr 2003, http://www-isd.fnal.gov/gridftp-wg/draft/GridFTPRev3.htm.

[3] Autodesk Maya. http://www.autodesk.com/alias. Maya is a registered trademark of Autodesk, Inc.

[4] BitTorrent: http://www.bittorrent.org/protocol.html

[5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium,* pp. 13-24. Seattle, WA (August 2000).

[6] D. Bosio, et al. Next-Generation EU DataGrid Data Management Services. *Computing in High Energy Physics (CHEP 2003)*, La Jolla, California, March 24–28, 2003.

[7] J. Chapweske. Tree Hash Exchange Format (THEX) http://www.open-content.net/specs/draft-jchapweske-thex-02.html

[8] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of OSDI: Symposium on Operating Systems Design and Implementation* (2002).

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01).* Banff, Canada, Oct 2001.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS 2002)* (July 2002).

[11] P. Druschel and A. Rowstron. A. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of HotOS VIII*, pp. 75–80.

[12] D. Duchamp. A Toolkit Approach to Partially Connected Operation. In *Proc. of the USENIX Winter Conference*, pp. 305-318, Anaheim, California, Jan. 1997.

[13] K. Eshghi and H. K. Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. HP Labs Technical Report HPL-2005-30R1, http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.html

[14] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid Computing: Making the Global Infrastructure a Reality. *The Physiology of the Grid*, Wiley, 2003, pp. 217–249.

[15] K. Fu, M. Frans Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 181-196, Oct 2000.

[16] Val Henson. An Analysis of Compare-by-hash. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems* (HotOS IX), Lihue, Hawaii, May 2003, pp. 13-18.

[17] HP Utility Rendering Service: http://www.hpl.hp.com/SE3D/whitepaper-urs.pdf.

[18] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proc. of the 4th Usenix Conference on File and Storage Technologies (FAST)*, Dec 2005.

[19] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, pp. 91–106.

[20] R. Merkle. *Secrecy, authentication, and public key systems*, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., 1979.

[21] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-Bandwidth Network File System. In *Proc. of the 18th ACM Symposium on Operating Systems Principles.* Chateau Lake Louise, Banff, Canada (October 2001).

[22] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the General Track, 2004 USENIX Annual Technical Conference.*

[23] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (2002).

[24] A. Rowstron and P. Drushel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware).* Heidelberg, Germany (November 2001).

[25] M. Rubel. Easy Automated Snapshot-Style Backups with Rsync. http://www.mikerubel.org/computers/rsync_snapshots/

[26] Rsync: http://samba.anu.edu.au/rsync/

[27] SE3D: http://www.hpl.hp.com/se3d

[28] H. Stockinger et al. File and Object Replication in Data Grids. In *Proceedings of 10th IEEE Intl. Symp. on High Performance Distributed Computing*. 2001.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the ACM SIGCOMM 2001.* San Diego, CA (August 2001).

[30] Niraj Tolia, Michael Kozuch et al. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proc. of the General Track, USENIX 2003 Annual Technical Conference*, pp. 127-140.

[31] W. Watson III, Y. Chen, J. Chen, and W. Akers. Storage Manager and File Transfer Web Services, *Grid Computing–Making the Global Infrastructure a Reality*. Wiley, pp. 789-801.