Highly Scalable Video Content Distribution
Drew Westphal / CS 490
Yang Richard Yang

ABSTRACT

The recent explosion of demand for video and other network-intensive content has spawned slews of new web services, each with different and innovative approaches to satisfying their users. While the types of services offered are continually changing, the design challenges facing the architect of a highly distributed system are similar across many different applications. This is a three layered server system is designed to make iPhone incompatible video content available, on demand, for a large community of iPhone users. With this potential scale in mind, the system has been designed from the ground up to accommodate a large and unpredictable user base while maintaining overall system performance.

INTRODUCTION: Design Motivations and Influences

YouTube, Hulu, and Akamai are some of the best-known names on the web for the user-transparency with which their systems provide massive amounts of content to a wide user base. In 2006, before its 1.65 billion dollar buyout by Google, YouTube reported that, per day, over 65,000 videos were uploaded and 100 million were viewed. By 2009, YouTube had reportedly served over 6 billion total videos and it’s estimated that 24 hours of content are uploaded to YouTube every minute¹. Similarly, Hulu claims that, in 2010, they streamed 903 million videos monthly². Broader in scope, the Akamai Content Delivery Network faces (possibly amplified versions of) the same design challenges, tackling content distribution and resource use across 61,000 servers, which handle a significant percentage of the total traffic on the Net³.

With the introduction of smart phones, such as Apple’s iPhone, even more users have the ability to request on-demand content, and many services are as yet unable to support these new devices. YouTube supports the iPhone, but at the expense of maintaining its own cache of separate, iPhone compatible videos. Smaller content providers, not to mention individual website owners, do not have the time or expertise to convert their video content for retrieval on a new smart phone. The video distribution system designed for this project is intended to address that shortcoming, by making iPhone compatible versions of those previously inaccessible videos available, with low latency, to a user base that could grow to millions. This solution requires a complex, multi-layered server architecture.

By designing and implementing such a system from the ground up, this project gives clear insight into the challenges that face large-scale content distribution systems. Java was chosen as the development language because of its strong networking capabilities and extensive standard API, which, together, facilitated a focus on high network and CPU performance as well as the rapid development cycle needed for this project.

¹ http://en.wikipedia.org/wiki/YouTube
² http://en.wikipedia.org/wiki/Hulu
HIGH LEVEL SYSTEM ARCHITECTURE

In order to quickly provide iPhone encoded video objects to a large number of users, the system makes use of three different types of servers. Multiple instances of each server type can be created in order to avoid network, CPU, and memory bottlenecks. The general function of each server type is outlined below and some features may currently be incomplete in the implementation.

Encoding Layer. This layer provides the encoding functionality for the system. An EncodeServer is able to download multiple video resources and encode them for the iPhone in parallel. The encoder is configured to provide relatively high quality videos at a bitrate below 160 kbps, which makes videos streamable over the Edge network, the lowest bandwidth portion of AT&T’s data infrastructure in use by the iPhone. The EncodeServer, along with processing external video resources, acts as a primary cache for those resources for other servers. The number of threads in use is configurable, and the EncodeServer is configured to monitor processor load, which can be used as a first approximation of the number of allowable simultaneous encodes. The EncodeServer does not communicate directly with the clients of the system (although it has that capability), but rather provides encoded videos to higher-level CacheServers, which are configured to rapidly distribute those videos across the network.

The system is designed to accommodate a large number of requests on machines whose configurations can vary widely. Each EncodeServer could potentially have different abilities to map outgoing files in memory (for faster I/O) or store files on disk. As such, the EncodeServer maintains a configurable LRU cache of on-disk files, which automatically purges those that are least requested once the size of the cache crosses the configured threshold. The caching system also manages the number of files that are mapped into memory for rapid output.

Caching Layer. CacheServers pull requested video content from the EncodeServers and cache it for rapid and repeated retrieval. A CacheServer is essentially a webserver that is optimized for sending and storing video content. Though the video resources served by this layer are the same as those generated on the EncodeServers, the division of labor between the two layers allows for higher potential performance. Both encoding video in parallel and running a high performance webserver place demands on CPU, memory, and I/O systems. This architecture’s intent is to more fully utilize system resources on each individual server, by using all system resources for one task. The multi-layered architecture allows for more flexibility and scalability. If the system’s use tends to focus on a few highly requested videos, fewer EncodeServers are needed. If, however, the system is used for a wide variety of content, more EncodeServers will be in use.

The CacheServer makes more extensive use of the caching system described in place on each EncodeServer. The intent is to use the multi-layered nature of the caches to maximize the likelihood that a client will be able to receive their video from a low-latency source. As the CacheServers are dedicated to sending out large files over the network, a majority of their physical memory can be allocated for the automatic memory mapping and unmapping provided for by the caching subsystem. The files that remain memory mapped is once again determined by least recently used order, with the hope that the majority of I/O transactions by the cache server will not need to incur costly disk reads. The static nature of the files served saves the system from the problem of worrying about disk writes and cache invalidation.
This caching architecture is apparently also the architecture in use by YouTube, whose implementation details are proprietary. It seems that there are compelling reasons for separating the encoding and distributing work, possibly the ones I’ve outlined above. Personal experience has demonstrated for me that a machine saturated with asynchronous client connections can do little else but serve them. A machine dedicated to encoding—especially in this low-latency environment—should have as little competition for resources as possible.

Load Balancing Layer. This is the layer that users of the system will encounter first, and its goal is to direct clients and lower layer servers such that system resources are used optimally. The most common type of load balancer is a Layer-4 load balancer, which is content blind and distributes requests according to load only. It provides failover detection and is sufficient for many systems. This application, however, would perform poorly with traditional load balancing as the requested video resource may exist on only one CacheServer. Sending a request to a server than one on which was already hosted would require for the video to be transmitted to that server, or worse, re-encoded, adding significant latency and bandwidth costs. Coordinating such transfers would also introduce a serious penalty in system complexity, as every cache server would need to be able to find any others that had the resource.

So, in the interests of performance, this system implements a basic Layer-7 load balancer, which is aware of application demands and can send requests directly to the lowest-loaded cache server on which that resource exists. If no such resource exists, the lowest loaded server is selected to service the request. Centralizing knowledge about the whereabouts of system resources in the load balancer decreases overall system complexity, as CacheServers and EncodeServers do not need to maintain records of resource locations throughout the system.

The load balancer performs a regular “heartbeat” style ping of the servers in the system as well as tables tracking resource distribution. In the case that a video becomes very popular and load for the CacheServers where it is stored becomes too high, the LoadBalancer is able to direct a client request for that resource to a new CacheServer, which will then initiate transfer of the primary copy and begin to serve that resource. This behavior allows the system to automatically account for highly requested videos by spreading them out, based on load, across a number of CacheServers.

Due to the nature of a TCP connection, most load balancers must act as a relay for responses from the servers for which they are balancing load, as the responding server does not share IP and port numbers with the load balancer. Given the size of the resources that this system is intended to transmit, as well as its intended scale, rewriting such a volume of response packets in the load balancer is not practical. Ideally, the system would implement a form of direct reply load balancing, which allows multiple servers to “share” the same IP, thus receiving a request from the load balancer and responding without the help of intermediary. For the first version of this service, a solution with near-identical server load characteristics, but less grace and technical complexity was implemented. The LoadBalancer responds to requests based on its table of available resources with a HTTP 302 Found message, which causes the client to connect directly to the CacheServer specified by the LoadBalancer. In a production environment, this solution might pose a security risk, as a sufficient number of well-designed requests could, essentially, reveal much of the system’s internal layout to an unauthorized party. For the purposes of an initial product, it is very similar to the direct-reply load balancing solution.
This solution is one of the more complex load balancing solutions available to applications, as it requires very site-specific configuration. It is well suited to the task and offers more for this specific application than even Amazon ELB would.

SYSTEM IMPLEMENTATION

Each protocol and subsystem of this server was written from scratch in order to gain a better understanding of the functionality of such a large-scale system. There are nearly 120 classes and interfaces that make up the final submission, which are not all covered here. The system has extensive online documentation in the JavaDoc format. Below, find an overview of the major protocols and classes involved in the implementation of these servers.

Client-Server Communication. Communication by clients with the LoadBalancer and CacheServers (and with the EncodeServers when the “–clients y” flag, value pair is set) takes place using the HTTP protocol. The LoadBalancer is queried with a GET request for the entire URL for the video to be encoded. The LoadBalancer responds with a 302 redirect to a CacheServer with the name of the resource as it is cached on that server (basically the file referenced in the video must have the proper extension to play properly). An example exchange for an already cached file is below.

```
Client  GET http://loadbalancer.com/movie.avi HTTP/1.1
LoadBalancer HTTP/1.0 302 Found
Location: http://cacheserver.com/movie.mp4
Client  GET http://cacheserver.com/movie.mp4 HTTP/1.1
CacheServer HTTP/1.0 200 OK
Content-Type: video/mp4
...
```

If the file has never been requested before, the client will be redirected to request the original resource name at the cache server and may be redirected a second time to request the properly named resource from that same CacheServer, once the name of that resource is known. This, however, is a very modifiable detail of the current implementation.

Server-Server Communication. Server-to-server communication takes place using the dew47.serialization package, which is modeled on the lightweight serialization in Apache Hadoop. Java’s built in serialization was not chosen for server-to-server communication because of its significant overhead and its poor suitability to asynchronous data streams.

The serialization package defines an ITransportable interface, which must be implemented by classes that will be serialized. The Serializer class provides a number of methods to write out and read in ITransportable objects from both synchronous and asynchronous data sources.

Each server in the system has a System-port for serialized communication, and a client port, for responding to client HTTP requests (the EncodeServer does not usually handle requests on its client port). EncodeServers and CacheServers must, upon startup, “squawk” in order to notify the LoadBalance server of their presence. After the squawk packet has been acknowledged, the servers will receive a heartbeat status ping on their system port at a configurable interval. Response to the ping indicates current load, as well as the resources that are currently cached on the given server. How the load balancer tracks this data will be covered in the next section.
CacheServers also use the System-port to contact the LoadBalancer in the event that they are queried for a resource that is not present. The LoadBalancer will direct the CacheServer to an EncodeServer from which the resource can be pulled—either from the local cache, or dynamically, after it is encoded.

**LoadBalancer Resource and Service Tracking.** The status response sent to the load balancer at every ping interval contains information on which ports (if any) are open to client communication, the location of the System communication port, as well a representation of load for that server (in this case, a double from 0 to 1). Finally, status responses contain a Java Set with an entry for every resource currently on the server that is reporting its status.

Java’s support for Sets makes it possible to easily track the changes in resource allocation upon the load balancer’s nodes. The LBServiceData class is delegated with tracking these changes and does so by maintaining a history of the two most recent resource-sets reported by the servers. Using these two sets, the asymmetric set differences indicate, on one hand, the resources that have been purged from the caches on the servers, and, on the other, the resources that have been added to the cache. The maps that pair resources with the service ports on which they can be accessed are updated according, and kept in a Java SortedSet (with the proper comparator) such that the lowest load server for a given resource can be retrieved with a single method call. A larger SortedSet of client and system facing resources is also kept, so that the lowest load CacheServer or EncodeServer may be chosen quickly.

Given that the goal of this system is to output streaming video, the load metric used for CacheServers is per client throughput on file writes as a percentage of the minimum target output bitrate for streaming (about 1.3 times the total video bandwidth). The dew47.LoadBalancer package contains a few interfaces that interact with the dispatcher threads to measure outgoing file size and total time to write outgoing files. Since not every part of the dispatcher should reasonably be modified to implement the IPerClientThroughputQueryable interface, the AsyncServer class, which aggregates per client throughput uses an instanceof check to only sum throughput from dispatchers which have been appropriately marked. The final throughput calculation is done via a moving median, which may provide a better indication of the underlying trend than a weighted moving average.⁴

**Asynchronous I/O.** The dew47.async package defines useful interfaces and classes for performing Asynchronous I/O. An abstract class, BasicDispatcher is extended to implement a dispatcher for accept requests and a separate dispatcher for read/write requests. The AsyncServer class chains AcceptDispatcher and ReadWriteDispatcher instances together in a one to many relationship to allow for multiple dispatcher threads without synchronization issues (the AcceptDispatcher passes accepted client sockets to the ReadWriteDispatchers in round robin order).

These dispatcher systems work with the IAcceptHandler and ISockReadWriteHandler interfaces to accept and read/write to sockets. A further layer of abstraction is added for the server programs, the IHTTPReadWriteHandler and IObjectReadWriteHandlers provide protocol specific handler hooks for HTTP request serving and serialized request handling. Use of these interfaces spares implementers any need to worry about the details of the asynchronous transactions. Every asynchronous server can have multiple IAcceptHandlers added to it, as well

as outgoing connections, with custom handlers. In all, this means that no machine needs to devote more than the number of threads used by their AsyncServer to any kind of Socket I/O. Keeping the thread count down is another way that this system aims for high performance.

Along with handlers for reading and writing, the ITimerInterruptible interface allows for threads to process an interrupt from a TimerThread, which can be used to implement TCP Flood protection, as well as unhealthiness after a LoadBalancer ping goes unresponded to.

Finally, asynchronous writing has been standardized by use of the INIOWriter interface, which allows all implementing classes to be written out to socket connections through one standard interface. This integrates well with the automatic memory caching system, which will be discussed next.

**File caching and memory mapping.** The system aims to manage system resources while maximizing output, which is why having the most frequently used files in memory is crucial to this application. The dew47.cache package implements a complex caching system that aims to address the caching needs of the EncodeServer and CacheServer. This class, called TwoTierCache, contains TwoTierManagedFiles. The TwoTier designation designates that there is a main cache of file entries on disk and a second tier cache of file entries that are memory mapped. Both of these internal caches as well as TwoTierCache implement the ICache< K,V > interface and store ICacheFile values.

MemoryCache and DiskCache are both descended from the SizeAwareLRUCache class, which defines a number of useful methods and abstract methods. DiskCache deletes entries that are removed from it when the size becomes too large, MemoryCache stops processing from acquiring references to the mapped byte buffer representing the memory mapped file. These file operations are made possible by a close integration with the TwoTierManagedFile class, which returns file handles and mapped byte buffers based on the containing caches policy. A checkout/checkin system for file handles and byte buffer references allows the TwoTierManagedFile to determine when it is appropriate to close file handles (to keep down the open file count) or to remove mappings from the MemoryCache.

The INIOWriter, mentioned above, encapsulates the handling of the TwoTierManagedFile. A TwoTierManagedFile will return an INIOWriter instance that will automatically write to a SocketChannel connection using the fastest available of the file handles at the time. Another INIOWriter implementation, the SpoolingINIOWriter allows for an incoming file to be used by multiple threads for output, and is aimed at decreasing latency.

The CacheServer uses the SpoolingINIOWriter when it spawns a thread (threads have higher per connection throughput than async I/O) to request a resource from an EncodeServer. As soon as the EncodeServer begins sending a response, the SpoolingINIOWriter is passed back to the client-writer thread and output begins. A cache entry is also created that points to the SpoolingINIOWriter, so that requests arriving at a CacheServer for a resource that is in transit to that server can be served to all incoming clients without delay. The incoming file is also reported to the LoadBalancer so that new requests are pointed to the CacheServer in question. In the background, the SpoolingINIOWriter is used by the CacheServer incoming file transfer thread to write the file out to disk and transparently swap the existing cache entry for a real TwoTierManagedFile cache entry.

**Video Encoding.** The video encoding package, dew47.encode, provides an EncodeQueue class with a configurable number of threads that accepts requests for encoding. While encoding is in
progress, multiple threads can request the same video resource, but the threads will all receive a reference to the same file. Video encoding is done by a fixed number of EncoderThreads, which try to keep their use of system resources low. Firstly, the EncoderThreads work with very lightweight EncodeRequest objects, containing only the bare minimum of information to properly prepare a request response. Threads are also reused, not respawned.

An encode request proceeds in a relatively straightforward manner. A buffered input stream is created from a URLStream pointing to the video resource. The URLStream is marked and then fed into an FFMpeg instance configured to output basic information about the video stream. The output text from FFMpeg is parsed to determine information about the incoming file and to adjust encode options (whether it's necessary to resample audio, adjust aspect ratio, etc), the stream is reset to mark and fed into an FFMpeg instance with the newly generated command string. The output is written to a file whose name is provided by a FileNameServer, which guarantees unique file names. This file is then added to the cache and sent out.

Very unfortunately, there is no encoder package that can write out an mpeg-4 (or other iPhone compatible) file without requiring random-seek access to the file. The trouble with the random-seek requirement is that it's therefore impossible to begin sending the file before it is written. This is the biggest flaw with my system—and I can do nothing to address it; although the encoder successfully reaches 100+ fps on many sources, the general latency on a first time encode is quite long.

CONCLUSION

Implementing a project of this scale has been an important learning experience for me. This was one of the first opportunities I've had to reap the true benefits of object-orientedness, simply because my system grew so large. Without the number of interfaces and abstract classes that I implemented, this project would have been even larger and its complexity would have been unmanageable. It is in this respect that I would first make changes. There are certain structural decisions about the design of my asynchronous handlers, especially, that were made too early to be effectively changed when I realized their weaknesses. Perhaps this is the double-edge to Object-Orientedness’ sword: that good design decisions bring many benefits, and bad ones make things very unpleasant.

As far as the overall structure of the system, I think that it is satisfactory, but it is very rough and skeletal in places. The protocol between the servers is bare and my serialization system, though very lightweight, slows development significantly and has been a major source for errors. Switching to java serialization, or a tested library solution that requires less hand coding of individual serialization methods would speed future development.

All of these problems considered, however, I still believe that I choose a relatively good, broad, and extensible design for this system. Even if I were to start over completely, there would be thousands of lines of useful code in the existing project, as well as a strong overall design that takes into account the important constraints of scalable, networked content distribution system.