Wireless Middleware: Dynamic Video Transcoding
Parikut Aghera Advait Dixit Ricardo Oliveira Vidyut Samanta
{parikut, advait, rveloso, vids}@cs.ucla.edu
Computer Science Department, UCLA

ABSTRACT
There are wide varieties of devices, with different screen resolution, color support and processing power, capable of streaming video from the Internet. To support all these devices, middleware proxy servers which transcode content to fit each client's device are becoming popular. In this paper we identify transcoding parameters (viz. frame size, color depth and Q-scale) that help in adapting to changing network conditions and computing resources on the proxy dynamically. We present an enhancement to three tier transcoding proxy architecture for streaming video, which enables dynamic changes in the these transcoding parameters while streaming.

Categories and Subject Descriptors
C.2.m [Computer-Communication Networks]: Miscellaneous

General Terms

Keywords
Dynamic video transcoding, middleware, network bandwidth, processor load, streaming video

1. INTRODUCTION
Today there are a wide range of devices with varying features such as screen size, color depth and processing power that connect to the Internet. Also they have different means by which they connect to the Internet, some may connect through wired 100Mbps connection, some may use CDMA wireless technology like cell phones, others using 802.11b, and in the near future 802.11g. Hence the effective bandwidth of each devices connection also varies. These conditions make it impossible for servers alone to provide QoS to all the clients. Hence proxy servers capable of transcoding content for various devices are becoming popular [1][2][3].

Video streaming is becoming very common today. Cell phones are equipped with cameras, and people can send videos taken on their cell phones to friends using MMS. With streaming data, there are two important points that should be considered:

1. For devices that are connected through wireless, their bandwidth changes dynamically, because of signal strength, and interference.
2. Access patterns of Internet data are highly unpredictable and can have spikes, which may be caused by a particular event. For example during a crisis a large number of clients will access streaming clips from a news website.

Hence, to provide QoS a video transcribing server should be able adapt dynamically when there are spikes, or when the bandwidth for a particular connection drops or increases. Current literature does not talk much about dynamic video transcoding servers; even less information is available on automatic video stream transcoding. In this paper we will discuss an implementation of a dynamic video transcoding proxy, which extends tinyproxy [5]. Implementation can be easily extended to make the dynamic transcoding decisions automated.

The following are some goals that we think are important for video transcoding:

1. Adapt the multimedia stream to fit client requirements. For example, frame size of the video is scaled to the size of the client’s screen.
2. Reduce latency experienced by user.
3. Perform these adjustments dynamically.
4. Provide a secure and scalable service.

Dynamic video transcoding cannot be substituted by multiple pre-stored formats of the content on the server. This is because we want to support changing parameters while streaming.

In section 2, we describe our general approach for dynamic video transcoding. In section 3, we describe the implementation that we used to validate our approach. Section 4 has experimental results which show the effect of different transcoding parameters on consumption of systems resources. We conclude in section 5.

2. OUR APPROACH
We identified the transcoding parameters of a video stream that can be changed to adapt the varying system resources. Section 2.1 describes these parameters and reasons for selecting them. Section 2.2 describes high-level architecture for changing these parameter dynamically.

2.1 Transcoding Parameters
After performing experiments, we came up with following three main parameters that should be changed dynamically:
2.1.1 Frame Size
This is the size of the video frame. It can be initially specified based on the clients screen size. In section 4, we have taken measurements of streaming video with different frame sizes and the results show that smaller frame size takes up less network bandwidth. If the client is experiencing bad network bandwidth while streaming a video, the parameter on the transcoding proxy can be dynamically changed so that the size of the frame is reduced and the client can continue to enjoy continuous streaming, but with smaller frame size. One could extend the proxy further to support handoffs between devices in which case also this dynamic scaling of the frame size would be very useful if the devices use different screen sizes.

2.1.2 Color Depth
MPEG standard doesn’t allow an arbitrary pixel format. Instead, each pixel is represented in the YUV scheme, where Y is the luminance and U and V the chrominance. Because each pixel is represented always with 12 bits (in the most used 4:2:0 planar format), we are only expecting minimal performance improvements by changing the video to gray scale.

2.1.3 Q-scale
The Q-scale is a parameter used to compress a given block of a video frame. Each block is a 8x8 pixel square to which a DCT (Discrete Cosine Transform), similar to JPEG compression mechanism, is applied. The coefficients of each block are then divided by a quantization matrix:

\[
C'_{i,j} = \frac{C_{i,j}}{qM_{i,j}} \quad (1)
\]

Where \( M_{i,j} \) is quantization matrix, \( C_{i,j} \) are the coefficients of the DCT applied to the image block and \( q \) is the Q-scale. The encoder discards those coefficients that are bellow a certain threshold, achieving a lossy compression this way.

![Figure 1: Transcoding operation with a high Q-scale](image)

2.2 Architecture
The transcoding proxy forms the middle tier of a three-tier architecture as shown in Figure 2. We added two components (shaded) that communicate with the transcoding proxy to enable dynamic modification of transcoding parameters. The transcoding proxy decodes the stream from a video streaming server one frame at a time; it then encodes the frame based on client requirements and parameters that can be changed on the fly. The new encoded frame is then streamed to the client. This all takes place in real-time.

2.2.1 Dynamic Transcoding Request Handler
The handler component, that accepts requests for change in transcoding parameters, is a part of the proxy. This change in transcoding parameters is passed on to the transcoder component of the proxy.

2.2.2 Dynamic Transcoding Request Generator
The generator component is responsible for generating transcoding parameter changes. This generator may be a part of transcoding proxy itself or a part of the client application or both. The generator can have added intelligence to make automated decisions depending on the client’s bandwidth and proxy’s resource utilization.

3. IMPLEMENTATION
We used a modified version of tinyproxy for the implementation. Tinyproxy is a light-weight http proxy server licensed under GPL [10]. We used a modified version of tinyproxy which had limited dynamic transcoding functionality. This implementation was able to change the Q-scale parameter of an MPEG stream to a hard-coded value at a fixed frame in the stream. We used the libraries, libavcodec and libavformat from ffmpeg [4]. The libavcodec library has audio/video encoding and decoding functionality and libavformat is a collection of parsers and generators for various audio/video formats.

3.1 Structure of the Transcoding Proxy
As shown in Figure 3, the transcoding proxy has one parent process which spawns multiple http server processes. Each of these server processes correspond to the transcoder of the architecture described in Section 2.2. The processes listen for a connection on a well know port and
serve incoming http requests. The parent process keeps track of number of free server processes and spawns additional servers if number drops below a threshold specified in configuration file.

The parent spawns an additional child which server as the dynamic transcoding request handler described in Section 2.2.1. The dynamic transcoding request generator is a stand-alone application which communicates with the request handler using TCP.

The server process transcodes one frame at a time as shown in Figure 5. First, the server process reads a frame from its input buffer and decodes it. Then, it reads the transcoding parameters in the shared memory. If they have been changed, it updates its codec. Finally, it encodes the frame and stores it in its output buffer. The rate at which the server process transcodes frames, is controlled by the rate control algorithm.

Figure 3: The transcoding proxy processes

Inter-process communication between the transcoding proxy processes is achieved using shared memory. Processes use locks to ensure atomic operation in shared memory.

3.2 Operation of the Transcoding Proxy

As mentioned earlier, all server processes are listening on a well-known port. The message exchanges at the transcoding proxy are shown in Figure 4. The client sends a request to this port. One of the server processes picks up this request. The request contains the address and port of the content server as well as the file required on that server. The transcoding proxy sends an http request to the content server. In response, the content server begins streaming the requested file. The frames of the stream are stored in the input buffer of the server process at the proxy.

Figure 4: Message exchanges at the transcoding proxy

Content Server Transcoding Proxy Client

3.2.1 Rate Control Algorithm

There was no control over the speed at which the transcoding was being done at the server. When the server was lightly loaded, the video stream was being transcoded at a much higher rate than the frame rate of the video. So, the effect of change in transcoding parameters was not reflected instantaneously. To solve this problem, we introduced a busy waiting scheme to control the rate at which each frame was being coded. This slows down the transcoding rate to minimize the difference between the frame rate of the video and the rate of transcoding of frames. Below is the pseudo code of the rate control algorithm:

Figure 5: Transcoding algorithm
last_time = get_system_time();
frame_interval = 1/Frame_rate;
if (a new video frame need to be encoded)
    current_time = get_system_time();
    delta = frame_interval * (current_time - last_time);
    if (delta > 0)
        busy_wait(delta);
    end if
end if

3.2.2 Implementation of Dynamic Transcoding Request Handler
The handler is implemented as a child process of the transcoding proxy which listens for transcoding parameter change requests on well know port (different from the transcoding proxy’s port for http). The handler executes the command specified in the requests and if required updates transcoding parameters of a particular http server process. Following are the commands are implemented in our handler implementation

gelist: The handler sends a list of currently active connections to generator along with the id for http server process that serves the connection.

update: The handler updates the transcoding parameters for specified http server process. Syntax for the update command is specified below.
update <http server process id>?<param_name=value>[&<param_name=value> ...
example: "update 1?q=30"

3.2.3 Implementation of Dynamic Transcoding Request Generator
The generator is implemented as a stand alone TCP client application which sends the transcoding requests to the handler, waits for the response and prints response from the handler on terminal. Following is formatted output of sample run with getlist command.

<table>
<thead>
<tr>
<th>Server Process ID</th>
<th>Client IP</th>
<th>Client port</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>131.179.224.88</td>
<td>34080</td>
</tr>
<tr>
<td>4</td>
<td>131.179.224.193</td>
<td>51354</td>
</tr>
</tbody>
</table>

4. EXPERIMENTAL RESULTS
We tested our implementation by measuring:

1) CPU usage,
2) bandwidth usage and
3) energy consumption

with different transcoding parameters(Q-scale, video frame size and gray/color scale). We used FFserver [4] as our content server and configured it to run at the same machine as the proxy. Thus, all our tests were done with the content server and the proxy located in the same physical machine:

```
Figure 6: Experimental setup
```

All the CPU measurements were done at the server machine and throughput measures were done taken from the client machine. At the client side, we used MPlayer [11] to play the video streams from the server. We turned off the rate control algorithm during the tests because we didn’t want to measure the busy wait cycles during the CPU benchmarks.

CPU usage was computed as the CPU time spent by the transcoder during the entire video stream. This CPU time has two components: system time and user time. The user time is the CPU time allocated to the main user-level process and all its children, while the system usage is the CPU time allocated to kernel processes (i.e., disk I/O, network access) during the user process execution. We used the C function getrusage() to return the values of CPU usage.

We used Tcpdump [12] to trace the downstream TCP connections at the client side and analyzed these trace files using Tcptrace [13]. Tcptrace gives several statistics about a given TCP connection, including the overall throughput of the connection.

We also did energy measurements at the server machine. In order to do these measurements, we used a laptop with a 1.8GHz AMD processor and 512MB of RAM configured as the server (a desktop was acting as the client). This kind of measurements can be affected by other processes running in the system at the same time. To avoid this interference, we turned off all unnecessary processes leaving only ffserver, tinyproxy and some kernel processes running. We used ACPI [14] to get readings of current voltage level and current drawn from the laptop battery. ACPI can be integrated in the Linux kernel and has a mapping to the /proc file system where current battery status can be accessed. ACPI takes measurements directly from the battery. So, we unplugged the power supply to avoid the constant recharging of the battery. In order to estimate the energy consumed by the transcoding proxy while processing the video stream, we took periodic measurements of voltage level and current intensity from the battery. We then computed the energy using also the CPU total usage time as follows:

\[ E = V \cdot I \cdot dT \] (2)

Where \( V \) is the average voltage and \( I \) is the average current being drawn from the battery in the time interval \( dT \).
4.1 Q-scale Measurements

For the CPU measurements with different Q-scales we used a 4 minute video clip file in avi format [15] (the video stream was encoded in MPEG-2 [17]). We took two measurements for each allowed value of the Q-scale (1-31) and averaged them. The results are shown in the next graph.

![CPU usage for different Q-scales graph](image)

**Figure 7** – CPU usage for different values of Q-scale.

As can be seen from the graph above, the total CPU usage has a linear decay for Q-scale<5 and a descendent trend as we go up in the Q-scale. This was expected since MPEG encoding performs a lossy compression over the video image and the amount of compression/loss is given by the Q-scale parameter. As the value of Q-scale increases, the quantization matrix will be reduced and will contain less relevant components, reducing the computation time for each block. As also can be seen from the above graph, the total CPU time is almost the same as the CPU user time, which means that the CPU system time is approximately constant and typically an order of magnitude lower than user time. The irregularity of the curves (ups and downs) is due to the fact that we were using a precision of seconds to measure the CPU time, which introduce some level of noise in the measurements (at least one second error margin).

The throughput measurements were taken using the same file as above with also two samples per Q-scale:

![Throughput for different Q-scales](image)

**Figure 8** – Throughput for different Q-scales.

The graph above clearly shows that throughput is reduced in a linear fashion for Q<5 and then converges to a stable value as we increase the Q-scale. This was also expected since as the Q-scale increases, the size of the blocks of each video frame is reduced, reducing the amount of data to be transmitted over the network.

4.2 Frame Scale Measurements

MPlayer uses avi decoder functions supplied by the libavcodec library. During our implementation we realized that libavcodec decoder used by MPlayer didn’t support frame size changes during the playback of the video stream. To overcome this limitation we used an image resize function provided by the libavformat library. This function basically resizes the video image and puts black strips around the resized image (in the case of a size reduction) but keeps the original height and width of the video frame:

![Frame scaling by 50%](image)

**Figure 9** – Frame scaling by 50%.

Black blocks have a high degree of compression (~100%) due to the MPEG image compression algorithms (similar to JPEG). In addition we can explore MPEG motion prediction algorithm to increase the compression level of the video stream and obtain both CPU and bandwidth savings.
The explanation for the throughput evolution with increasing value of frame scaling is the same as that for CPU.

4.3 Gray Scale Measurements

The MPEG standard uses the YUV scheme instead of the traditional RGB scheme to encode each pixel. Y is the luminance component, while U and V are the blue and green chrominance respectively. The most common format is the YUV 4:2:0 planar format that uses 3 planes (Y,U,V) to encode a given frame. By using gray scale, we are effectively using only the Y plane, since we are discarding the color information. However MPEG standards demand that all three YUV planes exist in the video stream. So in gray scale, each pixel is still represented by 12-bit YUV as in the color case. Most of the image information is in the luminance plane (Y plane). So, we do not achieve significant improvement in throughput and CPU usage by using gray scale, as can be seen in the graphs below.
4.4 Energy Measurements
We did several energy measurements under the conditions described previously. We measured the average voltage as well as the average current drawn from the battery during the video streaming transcoding. We changed one transcoding parameter at a time, so that we could isolate the effect of it on energy consumption. We were not interested in the absolute values of the measures but on the relationship between these values. So these measurements can also extend to other systems like desktops and server farms [8][16]. The reason we did this experiments at the server side vs. client side was because there is a great concern nowadays about optimizing energy consumption in data centers and content distribution networks.

![Figure 14: Energy consumption for different transcoder parameters.](image)

Energy was computed here according to Equation 2. We used a 4-minute video for each measurement. The energy measurements show a behavior similar to that of CPU usage for different transcoding parameters. We can see that great savings can be achieved by using the frame scale reduction or using high Q-scale values.

4.5 Capacity Testing
We wanted to find out how the server would scale with an increasing number of clients, each one downloading a video stream. We did CPU and throughput measurements with a variable number of clients connected to the server. We did these measurements at the transcoding proxy-side for the same reasons stated above for the energy. We were not doing any kind of transcoding during these measurements. The transcoding proxy decodes and encode the video stream even if no transcoding is necessary. The server being tested had the following specs:
- Dual Intel(R) Xeon(TM) CPU 3.06GHz
- 512 KB of cache size
- 4GB of RAM

![Figure 15: Cpu usage for different number of clients](image)

As can be seen from the graph above, the CPU usage is linear with the number of clients.

![Figure 16: Aggregate throughput for different number of clients](image)

These measurements were taken using the local loopback of the server (reason for the high values achieved). The throughput scales well up to 5 clients. After that it begins to saturate.

5. CONCLUSION
We identified transcoding parameters for video streams that can be changed dynamically to adapt the transcoding proxy to varying network conditions and computing resources on the proxy. We added two components - dynamic transcoding request handler and dynamic transcoding request generator - to the three tier architecture of proxy to perform dynamic change in transcoding parameters. Our measurements show that by changing the appropriate transcoding parameters, we can adapt the CPU usage of the transcoding process, as well as the amount of available bandwidth used by the clients. In a similar fashion, different kind of users with different device capabilities can take advantage of this dynamic adaptation at the server.

The next logical step would be to develop an automatic dynamic transcoder that could change the transcoding parameters without any user intervention. This would
require constant monitoring of available CPU and network resources.

6. ACKNOWLEDGEMENTS
We thank Prof. Songwu Lu for his guidance. We are thankful to Petros Zerfos for being our mentor for the project. Our thanks also goes out to James who modified the version of tinyproxy that we used.

7. REFERENCES