Client Level Framework for Parallel Downloading of Large File Systems

G.Narasinga Rao
Asst.Professor Dept.of CSE
GMR Institute of Technology
RAJAM-532127, A.P, India.
Srikakulam Dist

Srinivasan Nagaraj
Asst.Professor Dept.of CSE
GMR Institute of Technology
RAJAM-532127 .AP.India.
Srikakulam Dist

ABSTRACT
In this paper, Mirror sites enable client requests to be serviced by any of a number of servers, reducing load at individual servers and dispersing network load. Typically a client requests service from a single mirror site. We suggest a way for the client to access a file from multiple mirror sites in parallel to speed up the download. We have developed a technique that can deliver dramatic speedups as well as fault tolerance. Our approach doesn't require a feedback from the client to the servers, thus speeding up the process even more.

Keywords
Parallel downloading, Large file system, FEC Codes.

1. INTRODUCTION

Downloading a large file from a heavily loaded server or through a highly congested link can be a painfully slow experience. The many proposed solutions for addressing these problems share a common theme: improve performance at the bottleneck. For modem users, there is not much that can be done: to improve downloading time they must either upgrade to higher baud rates or settle for receiving distilled lower bandwidth versions of the content they wish to access. However even today, not all modems run at full speed due to network and servers loads, and much can be done to solve this problem as well.
For most of us, for whom the last mile is not the bottleneck, there are a wide variety of techniques to improve performance in the network and at the server. The most relevant to
our discussion is the use of mirror sites. The mirroring approach deploys multiple servers storing the same data at geographically distributed locations, in an effort to both distribute the load of requests across servers and to make network connections shorter in length, thereby reducing network traffic. A limitation of current mirroring technology is that the user must choose a single mirror site from which to access data. While the choice of server may appear obvious when the number of mirror sites is small, some works indicate that the obvious choice is not always the best choice and dramatic performance improvements can result from more careful selection.

Our first objective is to enable users to download data from multiple mirror sites in parallel in order to reduce downloading time. This technique not only has the potential to improve performance substantially over a single server approach, but can eliminate the need for a complex selection process. We do so, by using forward error correction codes, as described below.

2. FORWARD ERROR CORRECTION CODES

FEC techniques are generally based on the use of error detection and correction codes. These codes have been studied for a long time and are widely used in many fields of information processing, particularly in telecommunications systems. In the context of computer communications, error detection is generally provided by the lower protocol layers, which use checksums (such as CRC) to discard corrupted packets.FEC codes were designed to allow recovery of the original data from the packets, which have arrived. FEC codes can be also extended to allow reception of data from multiple sources.As mentioned above, Forward Error Correction (FEC) codes allow a recovery of data sent over an unreliable channel, where data packets can be received incorrectly or even lost. Sending a redundant data, which is sent along with the original data, does this. If the size of our data is k packets, FEC codes encode the data in such manner that the original data can be reconstructed from any k packets received. Multiple servers can send these packets to the receiver. A receiver could gather an encoded file in parallel, from multiple sources .As soon as any k packets arrive from any combination of the sources, the original file can be reconstructed.

The scenario of a file download process should be as follows:

The user wants to download a file, either via a hyperlink on a web page, or directly (the user has the URL). The user then clicks on the hyperlink, or enters the URL in the browser window. This takes the user to an HTML page, containing a Java applet, which is responsible to the download process. The applet downloads the file, saves it to user's hard drive, and optionally opens it in the browser window (if the browser can display this media type).

3. OUR SYSTEM CONSISTS OF SEVERAL PARTS

A distribution center, residing on a dedicated machine (either as a server or as a local application). This application will receive the file from the file creator; encode it using the FEC algorithm producing 's' encoded files, when 's' is the number of data servers (described below). The original file is divided into several chunks, and each chunk is encoded separately. The distribution application will upload the encoded chunks to the data servers. Each server will hold a full image of the original file, so the file can be downloaded even when there is only one data server available. The distribution center can work with up to 8 file servers (imposed by the limitations of the current implementation of our FEC algorithm). Data servers, then hold the encoded file chunks. The client will download the file from these servers using the HTTP protocol.The client is a Java applet, which manages the download process. The applet will initially download the data server list, containing the locations of all data chunks. Then, it will download and decode the chunks in the original chunk order. While the first chunk is being decoded, the second chunk will be downloaded in parallel, and so on. Finally, all the chunks are combined to reproduce the original file. The whole process is completely transparent to the user, who will only see a progress bar and a completion notification.

The distribution server

The distribution server will allow the content distributor to encode the content and upload it to a predefined group of servers, which will be specified as a configuration file. Each chunk of the encoded data will be stored in a separate file, since this will make the download manager implementation simpler. Also not all servers allow downloading from an arbitrary position in the file (e.g. some proxy servers don't support it). The names of the encoded files will be determined by the application based on the original file name. As mentioned above, there is a limit of 8 servers at this moment.

The distribution application will also prepare the HTML page that contains the client applet, which will manage the download process on the client machine. The HTML file will also have an embedded data regarding the locations of the data (the list of the mirror sites). This data

will be passed to the applet when the applet is activated.

The download manager

The user can start the file download either by following a link from a web page or by entering the URL of the download manager (the HTML page). When the user initializes the download process (by either method) it receives the HTML page, which was prepared (customized) by the distribution application for this particular file.

The HTML page contains a Java applet, which is the download manager and is described below. First of all, the Java applet will parse the HTML page and extract the locations of the mirror sites. Then, we open HTTP connections to all the mirror sites, and start downloading the first chunk of data from all mirror sites in parallel. Now we wait until the download of the first chunk finishes, and then we start downloading the second chunk and decoding the first chunk in parallel. When the last chunk finishes downloading the user must wait until the decoding of the last chunk is over. This is the only time when the user actually feels the price of the decoding process. We try to minimize this time by choosing a smaller chunk size, although it is negligible when using fast CPUs and JVMs that support Just In Time compiling (JIT). When the process finishes the downloaded file will be saved to the disk.

How a chunk is being downloaded?

We receive the data from all the servers and as soon as we have enough data for the chunk we abort all existing connections. In order to avoid unnecessary packets to be sent over the network, we close all but the fastest connections when the amount of data we have already received is close enough to the chunk size. We close the last connection as soon as the last byte needed for the decoding arrives. However due to the TCP limitations more unnecessary data can be received (due to a large window size or fast network).

Since the opening of a new connection to a remote server (the three way handshake and sending and processing the request header), we open the connections for the next chunk before the current chunk is completely downloaded .The amount of data sent during the setup is not big and we save precious time which otherwise would be wasted.If a connection to one of the servers fails, we try to open another connection to that server, and resume downloading from the position we stopped. If resume is not available, we might consider starting from the beginning, or give up using this server for the current chunk, depending on the amount of data we have already downloaded from this server, and the download progress on other connections.

The data format

Our basic unit of work is a ***packet*** (1KB). A ***strip*** is a sequence of packets, in our case, 32 packets. Therefore, the size of a strip is 32Kb. 32 strips are combined to form a ***chunk*** (1Mb). Strips are the basic units of the encoding/decoding process, while chunks are the basic units of the download process. We store each chunk of encoded data in a separate file.
According to our FEC algorithm, each server must hold a complete (encoded) copy of data. Therefore the total size of the encoded data is the size of the original data multiplied by the number of the data servers.

The encoding procedure

In the encoding procedure we take a file, and split it to chunks, and then split the chunks to strips. Let's focus on a single chunk. Let 'n' be the number of servers. The encoding algorithm produces 'n' encoded strips out of one original strip. Each encoded strip will go to a different server. Then for each server, we collect all its strips and store them in an interleaved format described above. For each packet we must know few things for the decoding procedure that will occur at the client:

1. The strip it belongs to.
2. Its index in the strip.
3. The server on which the strip resides.

We don't have to specify these details explicitly for each packet. The client knows exactly, which server the packet came from, and the strip number can be calculated from the relative position of the packet in the chunk.

The decoding procedure

The decoding procedure reconstructs a strip from a collection of packets belonging to that strip. As we mentioned above, if the size of a strip is 'k' packets, then any 'k' packets from any server in any order can be used to reconstruct the strip (of course only if they belong to that strip). After all strips are decoded, we combine them back into a chunk, and append the chunk to the target file. When the last chunk is written, the download process is complete.

The FEC Driver API

This API provides an application, the ability to encode and decode data using the Forward Error Correction (FEC) Code implementation based on the Vandermounde matrices. We have implemented the following classes:

FEC_Math:

This class includes various math functions necessary for handling matrix algebra over prime fields, which is necessary to speed up the work of the FEC core routines.

FEC:

This class contains the core FEC encoding/decoding routines. An object of this class can be created based on the following parameters: the number of packets in the strip (k) and the number of encoded packets (which in our case is a multiple of k).

FEC_Driver:

This class implements the required API (the encode and the decode routines), and provides our implementation constants – the packet size, the strips size and the chunk size. It also provides an interface to asynchronously query the FEC driver about its progress on encoding and decoding operations.

Design Description

FEC API: Using the API in order to encode the chunks.

Parser: Parsing the server configuration file.

ParserHtml Preparing the html file, this will consist of the download manager applet.

FTP: A class which gets a specific server configuration and uploads a chunk to this specific server using FTP. The class extends the Thread class and can be run in parallel with other tasks.

GuiFrame: The main frame of the application. This frame provides two file dialog boxes enabling the user to choose the file to distribute, and the server configuration file. It also provides a button that actually starts the procedure and a progress bar and a status line informing the user about the progress of the procedure. When the "start" button is pressed the a thread that will do all the encoding and uploading procedure is started.

Encode File: This classs (run as a thread) is responsible for the distribution process. It gets the file to encode and the server configuration file from the main frame (GuiFrame).

It first extracts the server information (Parser) from the server configuration file. When start is pressed in the GuiFrame the thread starts to process the source file chunk after chunk. After encoding (using the FEC API), it opens threads (FTP) to handle the uploading procedure to the servers.

4. SIMULATION RESULTS

Below there are the results of our simulation. In our simulation we have used 3 servers at Geocities, Acme city and Fortune city which provide free homepages for the public. We have uploaded the encoded file to these servers and then downloaded the file from these servers using our demonstration applet.

Server 1: acmecity			
	1MB	4.68M	KB/Sec
Test 1	103	1885.04	2.540943
Test 2	266	1385.28	3.465485
Test 3	263	1230.84	3.899386
Test 4	388	1815.84	2.63917
Test 5	218	1020.24	4.690724
Test 6	203	1324.44	3.611337
Test 7	277	1298.28	3.69875
stdev	67.2904	314.919	0.74197
avg	304	1422.72	3.550549

Server 2: geocities			
	1MB	4.68M	KB/Sec
Test 1	156	730.08	5.564103
Test 2	164	720.72	5.64935
Test 3	159	744.12	5.44025
Test 4	150	702	5.82666
Test 5	206	964.08	4.97087
Test 6	212	992.16	4.83018
Test 7	230	1076.4	4.45217
Stdev	33.6303	157.39	1.01745
Avg	181	847.08	5.21408

Server 3: fortunecity			
	1MB	4.68M	KB/Sec
Test 1	69	276.12	17.3659
Test 2	65	304.2	15.7538
Test 3	69	322.92	14.8405
Test 4	62	290.16	16.5157
Test 5	66	308.88	15.5157
Test 6	63	294.84	16.2539
Test 7	77	360.36	13.2537
Stdev	5.84318	27.3461	1.30625
Avg	68.8571	308.211	15.6477

Our algorithm			
	1MB	4.68M	KB/Sec
Test 1	32.0513	150	33.1496
Test 2	33.0744	159	31.2701
Test 3	38.8889	167	27.3186
Test 4	35.6838	157	29.7724
Test 5	32.547	157	31.5687
Test 6	49.1453	230	21.5173
Test 7	43.8034	205	24.2536
Stdev	5.79783	27.1338	3.92237
Avg	38.1563	178.571	25.4354

If we sum the average download rate of the single servers we get 24.97334KB/s which are close to the data transfer rate we receive in our application. The gain rates we achive in throughput are 712% compared to Acme city, 389% compared to Geocities and 81% compared to Fortune City.

5. BENEFITS OF OUR PAPER

In this section we will discuss the benefits of using our algorithm. The question why use our algorithm will probably arise since there are other ways to download data from multiple mirror sites. The most simple of them is to try downloading the file from one of the servers. If the download is slow, stop that download and try another (hopefully) faster mirror site. Some programs such as Get Right try to figure out which server has the highest chances to be fast and download from that server. However, if the program is wrong we are stuck with a slow server. Another approach would be to split the file into chunks and download every chunk from another server. The problem with this approach is that if one of the servers is slow or down, we cannot receive the file. Our approach eliminates the difficulties of the methods described above.
The main advantages of our approach are:

1. Using our project we gain better throughput at the client side, thus gaining more speed than downloading from a single site. However this is true only when the bottleneck is at the server side and the client has available bandwidth that is not used due to the bottleneck at the server side. While using our algorithm we use more of the available bandwidth (since we open connections to several mirror sites) and therefore improving throughput when possible.

2. Fault tolerance – using our algorithm allows the user to download a file even if all the servers but one is down. As explained before every server holds a complete image of the file. Connecting to that sever will accomplish the task of downloading the file. (This is also true for the scenario when the server crashes during the download process). Using the methods described above when a server fails can result in a lost download if the servers do not support resuming.

3. Trying to minimize the amount of unnecessary data sent on the network. When using one of the methods above, if we choose not to use a server all the data we have received from that server is lost. In our case, even if a single kilobyte was received from the server it can be used. Of course this means that servers send less data, and therefore the load on the server's decrease that in turn reduces the chances of server crashing down.

4. The download is highly parallel – All the time we receive data from servers. Using the above methods is likely to end in waiting for a single server to send its part of data. Some connection can be very fast but the slowest connection will detain the whole downloading process. Using our algorithm does not have this effect since we download from all the servers all the time, until we have enough data to complete the download. Of course fast servers will contribute more to the downloaded file. No time wasted on waiting for "slow" servers.

6. CONCLUSION

In this work we have introduced a new way to speed up downloads from multiple mirror sites which dynamically adjust to the network and server loads. We have implemented the Forward Error Correcting Code from Luigi in the Java language and have built a Java applet which demonstrates our download technique. The impressive results that we got show that this technique has a high potential in achieving more speed from the current network infrastructure. Parallel downloading (PD) has been adopted recently in some Internet file downloading systems, and is expected to be more commonly adopted with the increasing deployment of CDN and peer-to-peer networks. The work reported in this paper was initiated by the lack of an in-depth analysis of the system performance and the impact on the whole system of such a popular scheme. It should be noticed that our conclusions are drawn based on a homogeneous network scenario and on the average downloading time. For heterogeneous scenarios where clients have different connectivity, average downloading time may not be a suitable performance metric to

study. The result shows that PD is not necessarily such a great scheme to adopt. When the servers are constrained in terms of number of concurrent sessions that they can serve, we have shown that admission control should be deployed to prevent unnecessary system degradation. We have presented in this paper that if the number of servers is limited, the system should limit the number of users regardless of the downloading scheme. This admission control process will prevent the average downloading time from rising without increasing the blocking rate.

REFERENCES

[1] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed Content Delivery Across Adaptive Overlay Networks. In Proc. of ACM SIGCOMM, 2002.

[2] J. W. Byers, M. Luby, and M. Mitzenmacher. Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads. In Proc. of INFOCOM, pages 275– 283, New York, NY, Mar. 1999.

[3] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In Proc. of ACM SIGCOMM, pages 56–67, Vancouver, Canada, 1998.

[4] R. L. Carter and M. Crovella. Server Selection Using Dynamic Path Characterization in Wide-Area Networks. In Proc. of INFOCOM, volume 3, pages 1014–1021, Kobe, Japan, Apr. 1997.

[5] Digital Fountain Inc. Digital Fountain's Meta-Content Technology. White Paper.

[6] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A Novel Server Selection Technique for Improving the ResponseTime of a Replicated Service. In Proc. of INFOCOM, volume 2, pages 783–791, 1998.

[7] L. Kleinrock. Queueing System, volume 2 (Computer Applications). JohnWiley and Sons, first edition, Apr. 1976.

[8] Y. Liu, W. Gong, and P. Shenoy. On the Impact of Concurrent Downloads. In 2001 Winter Simulation Conference, Arlington, VA, 2001.

[9] A. Myers, P. A. Dinda, and H. Zhang. Performance Characteristics of Mirror Servers on the Internet. In Proc. of INFOCOM, volume 1, pages 304–312, 1999.

[10] Y. Nebat and M. Sidi. Resequencing Considerations in Parallel Downloads. In Proc. of INFOCOM, 2002.