Controlled Transfer of Qclic Files

Project Status Report[1]

Krzysztof Zmudzinski

CS 737

for Prof. Miron Livny

Abstract

The purpose of this project was to develop an algorithm for controlled transfer of Qclic files. The algorithm controls the bandwidth and transfer time by delaying program execution or by changing the size of the transfer buffer. Two models were built to implement the algorithm: a pipe model and a TCP model. In both models, the algorithm performed similarly exhibiting the same properties. Delaying program execution results in almost perfect bandwidth and transfer time control but it also results in bursty traffic. Changing the size of the transfer buffer requires more work as it exhibits wide variations in the size of the buffer and often does not result in as tight control as its delay counterpart.

1. Introduction

Quality-controlled lossy image compression (Qclic) uses RD-OPT algorithm for optimizing JPEG quantization tables for individual images. Its goal is to meet any desired compression and/or quality specifications and to optimize the compression-quality tradeoff. For large sets of images, Qclic offers optimization of the compression-quality tradeoff for individual images as well as across a set of images. Such sets of images could be deposited on a network file server and requested by various clients, such as web browsers.

The purpose of this project was to develop an algorithm for controlled transfer of Qclic files. *Controlled* means that the user specifies what files should be transferred, how much bandwidth should be used to transfer these files, and how much time should the transfer take. The algorithm is responsible for transferring the specified files without exceeding the specified bandwidth and time. The algorithm takes advantage of progressive JPEG format and transfers files one scan at a time. However, this introduces problems for some web browsers (e.g. some versions of Netscape) which do not handle progressive JPEG images. In this project Internet Explorer, version 3.0 was used as it does handle progressive JPEG images.

The rest of this report consists of a description of the two implementation models in section 2, followed by a description of the algorithm in section 3. Section 4 presents some performance measurements and short evaluation of the results. Section 4 also presents existing problems with the algorithm and possible ways to fix them. Section 5 further expands on more possible ways to improve the algorithm and its performance.

---

[1] This report, the source code, and additional information are located at:
http://www.cs.wisc.edu/~zmudzin/737/737.html

2. Implementation Models

Two models were built to implement the algorithm: a pipe model and a TCP model. The pipe model was implemented as a simple client-server scheme that uses pipes to communicate between the client and the server. There are three pipes in this model:

- data pipe - used for actual file transfer;
- control pipe - used to send control commands to client;
- ack pipe - used to send acknowledgments of data transfers and control commands back to server.

Separate pipes for sending control and acknowledgments were needed because the server would often overrun the client and because pipes are one directional inter-process communication facilities.

The TCP model uses Berkeley sockets to communicate between the server and a client running on two different hosts. The only new issues that had to be addressed in this model were:

- passing arguments to the server (file names, bandwidth, time, etc.);
- the number of sockets (as opposed to the number of pipes in the pipe model).

There is only one socket needed in this model as TCP handles flow control and sockets are two-directional IPC facilities.

The following are formats for packets used in this model to send requests and to transfer files:
(each box represents a long (32-bit) value)

- Transfer request packet. Sent from client to server to request a file transfer with the included properties. After a transfer request packet, server will be expecting at least one file request packet.

```
+----------------+----------------+----------------+----------------+
|  TRANSFER_REQ  |   bandwidth    | transfer time  | transfer mode  |
+----------------+----------------+----------------+----------------+
```

- File request packet. Sent from client to server to request that specified file be included in the transfer. Length is length of file name and type is used to let the server know that this is the name of the last file requested. The file request packet is followed by a variable length packet containing the name of a file. File # is used as a handle for file buffer packets.

```
+----------------+----------------+----------------+----------------+
|    FILE_REQ    |      type      |     file #     |     length     |
+----------------+----------------+----------------+----------------+

+----...----+
| file name |
+----...----+
```

- File buffer packet. Sent from server to client with contents of the file described by file #. File # is first associated with a given file in file request packets sent by client. Length is length of file contents and type is used to let the client know whether file contents should be used to create the file described by file # or to append them to an existing file. The file buffer packet is followed by a variable length packet containing the file contents.

```
+----------------+----------------+----------------+----------------+
|    FILE_BUF    |      type      |     file #     |     length     |
+----------------+----------------+----------------+----------------+

+------...------+
| file contents |
+------...------+
```

- Transfer summary packet. Send from server to client to let the client know that the transfer was completed. In addition, the actual transfer size and transfer time (as seconds and micro seconds) are included.

```
+----------------+----------------+----------------+----------------+
| TRANSFER_QUIT  | transfer  size | transfer  time   (secs | usecs) |
+----------------+----------------+----------------+----------------+
```

3. Algorithm

There are two basic modes in which files are transferred:

- alternate - a file is opened, one scan is transferred, the file is closed, next file is opened, one scan is transferred, etc., the first is opened again, etc., and so on until we reach transferSize[2];

- sequential - transferSize is equally divided between all files; each file is opened only once and its share of transferSize is transferred.

Each of the two basic modes has three variations:

- exact - transferSize is really equal to bandwidth * transferTime; scans may be transferred partially. This variation results in the exact number of bytes transferred as requested by a client and in at most one file whose last scan was transferred partially for alternate mode and in all files whose last scans were transferred partially for sequential mode.. This variation is designed for clients able to handle progressive JPEG images even if scans are not complete.

- next scan - transferSize may be exceeded so scans are transferred in their entirety. This variation results in files whose scans are transferred in their entirety and with the actual number of bytes transferred possibly exceeding the transfer size requested. This variation is designed for clients able to handle progressive JPEG images but only if all scans are complete and that are willing to wait for those images longer than the requested transfer time.

- previous scan - transferSize may not be exceeded and scans must be transferred in their entirety. Similar to the next scan variation but transfer time will not be sacrificed resulting in one of files smaller by one scan as compared to the same file transferred using the next scan variation for alternate mode and possibly many smaller files for sequential mode since no file's share of transferSize will be exceeded.

The following are transfer modes which can be passed to the GetQclicFiles function:

- TM_ALT_EXACT - we alternate files and stop as soon as we reach transferSize regardless whether we read the entire scan of the current file or not;

- TM_ALT_NEXT_SCAN - we alternate files and stop once we finish reading the entire scan of the current file such that the total transferred size will be bigger or equal to transferSize;

- TM_ALT_PREV_SIZE - we alternate files and stop once we know that reading the entire scan of the current file would cause the total transferred size to be more than transferSize;

- TM_SEQ_EXACT - we divide transferSize equally among all files and transfer each file up to its share of transferSize; scans may be transferred partially;

[2] transferSize = bandwidth * transferTime

- TM_SEQ_NEXT_SCAN - we divide transferSize equally among all files and transfer each file up to and including the scan that will use the size allotted to that file and possibly (and likely) exceed it;

- TM_SEQ_PREV_SCAN - we divide transferSize equally among all files and transfer each file up to but excluding the scan that would cause the size allotted to that file to be exceeded.

There are two ways to control file transfer:

- time-out - after each send, we compare the time it should take to transfer the current total number of transferred bytes at the specified bandwidth to the actual time it took us to transfer those bytes and wait for the difference, if any. We calculate time to wait as follows:

  ```
  timeToWait = bytesTransferred / bandwidth – (currentTime – startTime)
  ```

- buffer size adjustment - we change the size of buffer used to transfer files to control the transfer rate; this solution has its drawbacks for low bandwidth and large files - more system calls have to be executed. This method requires more work as it exhibits wide variation in buffer sizes and often fails to control file transfers as well as its time-out counterpart. We calculate new buffer size as follows:

  ```
  bufferSize = (currentTime – startTime) * bandwidth * bufferSize / bytesTransferred
  ```

4. Performance

The time-out version of the algorithm exhibits good control over the actual bandwidth and transfer time. However, this is accomplished through a bursts-and-silences method. Although the overall bandwidth is almost identical to the requested one, the actual bandwidth measured in small intervals would very likely suffer wide degree of deviation from the goal bandwidth. This could be fixed by using small buffers to send data. This would however result in many more calls to send and many more waits in between calls to send.

The buffer size adjustment method was designed to fix the bursts-and-silence anomaly but in its current form it does not converge quickly to a constant buffer size. In fact, it suffers from frequent changes from the minimum to the maximum buffer sizes.

Table 1 shows results from running the algorithm with the time-out method for the TCP model and table 2 shows the results of the same tests but for the pipe model. The following numbers were obtained by running the same request three times and obtaining averages from the results.

Server was executed with the following arguments:
128 for the buffer size (in bytes),
1 (CM_TIMEOUT) for the time-out control mode.

Client was executed with the following arguments:
host for the host name where server is running,
CRBW for client requested bandwidth (in bytes/second),
10 for the requested transfer time (in seconds),
1 (TM_ALT_EXACT) for the alternate, exact transfer mode.

Pipe program (gqcf) was executed with the following arguments:
CRBW for bandwidth as above,
10 for transfer time as above,
1 for transfer mode as above,
4096 for the size of the client’s buffer (used for initial testing of problems with overrun pipes),
128 for the size of the server’s buffer,

1 for control mode as above.

The following are the abbreviations used in the tables:
CRBW - client requested bandwidth,
ABD - actual bandwidth,
ATT - actual transfer time,
MAX WT - maximum wait time between sends,
TOT WT - total wait time during transfer,
WAIT # - number of waits during transfer.

CRBW	ABD	ATT	MAX WT	TOT WT	WAIT #
1,000	999.65	10.00	1,289,66	9,399,657	77
2,000	1,999.34	10.00	682,65	8,840,527	149
3,000	2,998.77	10.00	469,68	8,440,959	221
4,000	3,998.55	10.00	386,44	7,769,190	285
5,000	4,994.60	10.01	331,32	7,251,270	342
6,000	5,995.30	10.01	299,73	6,826,215	402
7,000	6,994.52	10.01	251,94	6,410,627	451
8,000	7,990.02	10.01	212,35	5,971,018	503
9,000	8,754.43	10.28	206,05	4,870,278	482

Table 1: server 128 CM_TIMEOUT; client host CRBW 10 TM_ALT_EXACT

CRBW	ABD	ATT	MAX WT	TOT WT	WAIT #
1000	997.58	10.02	128782	9328155	77
2000	1997.31	10.01	68863	8759847	148
3000	2992.82	10.02	46350	8132732	214
4000	3982.84	10.04	36902	7657941	275
5000	4982.79	10.03	33045	7305481	343
6000	5979.44	10.03	26654	6902174	403
7000	6974.73	10.04	24974	6586655	467
8000	7973.79	10.03	20998	6206523	525
9000	8974.88	10.03	22945	5705130	558

Table 2: gqcf CBW 10 TM_ALT_EXACT 4096 128 CM_TIMEOUT

5. Possible Improvements

The buffer size adjustment mode needs more work. At this point it is not stable and the size of the buffer suffers a wide range of changes during a given file transfer which does not accomplish the goal of this version of the algorithm which is to keep the steady stream of bytes as opposed to having bursts and silences.

Another improvement would be to add another process to the server which would only be responsible for streaming data across the TCP connection to the client. That process would be “connected” by three pipes similar to those in the pipe model presented in this report. This would allow for streaming data across network while the main process is waiting for a disk operation to complete.

Yet another improvement would be to abandon TCP protocol and to implement our own that would allow for more control over how data are sent. This however, would force authors of web browsers to add another protocol to their programs. One point that has to be made about the current implementation of the algorithm is that it does not take into account TCP headers and IP headers sizes in calculating the actual bandwidth. It is assumed that when a client specifies bandwidth, it is bandwidth of the actual data transferred and our own packet headers.

The algorithm described in this report works fine for similar files (the same transferred size corresponds to the same quality). For files that are different, we could look at the quality and not the size when making decisions how much of each file should be transferred. The assumption in implementing this algorithm was that similar-size files would be transferred and Qclic algorithm to compress them would be run with similar parameters.

6. Summary

This report presents an algorithm for controlled transfer of Qclic files. The algorithm makes sure that the bandwidth and the transfer time specified by a client are not exceeded. It implements two different methods to accomplish its task: the time-out method and the buffer size adjustment method. Although the time-out method exhibits much better control than its counterpart, it results in more bursty traffic. On the other hand, the buffer size adjustment method does not work well as currently implemented and would require more tuning to assure quick convergence of buffer size.