ISSN:2231-0711

Lalit Adithya et al | IJCSET(www.ijcset.net) | December 2016 | Vol 6, Issue 12, 401-405

Segmented File Transfer

Lalit Adithya[#1], Sampada K S[#2]

[#]*Department of Computer Science and Engineering, RN Shetty Institute of Technology*
RN Shetty Institute of Technology,
Karnataka, Bangalore 098

***Abstract*— Of the many methods that can be used to download or upload a text or binary file to the internet, the method of Segmented File Transfer is one that is the lest applied even though the advantages of implementing the concept has a lot of advantages. This paper presents the advantages of using the concept of segmented file transfer while showing the necessary implementation details that are required to implement this concept for a download manager.**

***Keywords*— Segmented File Download, Byte Serving, Segmented File Upload**

I. INTRODUCTION

There is an increase in the number of active internet users by around 50% and there is an increase in the number of active desktop and laptop users by 64%. Even with this massive rise in the number of internet users, the average speed of these connections is 5.6 Mbps characterized with poor Quality of Service, that is, frequent disconnections resulting in data being lost or left corrupted.

Segmented file-transfer (also known as multisource file-transfer or swarming file-transfer) is defined as the coordinated transmission of a computer file sourced from multiple servers to a single destination. It can be applied as well when downloading the same file from the same server in various parts. A computer program downloads (retrieves) different portions of the file from various sources simultaneously, and assembles the file on the destination computer data storage device.

II. RELATED WORK

The File Transfer Protocol (FTP) is a standard network protocol used to transfer computer files between a client and server on a computer network. This protocol is built on a client server model and it uses a separate control and data connections between the client and the server [1]. Most web browsers today can retrieve files that are hosted on an FTP server.

When FTP was designed, it was not designed to be a secure protocol, and hence it has many security weaknesses [2]. In 1999, the authors of RFC 2577 stated that FTP was vulnerable to the following kinds of attacks:

- Brute force attack
- FTP bounce attack
- Packet capture
- Port stealing
- Spoofing attack
- Username enumeration

It is also known that FTP does not encrypt the traffic and all transmission are in clear text and therefore a file containing sensitive information such as passwords can be read by anyone who is performing sniffing on the network [3].

A. Hyper Text Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems [4]. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP is the protocol to exchange or transfer hypertext.

HTTP defines verbs that are used to indicate the desired action to be performed on a resource. The resource is usually a file that is residing on the web server. The may also correspond to the output of an executable file that is present on the web server. The HTTP/1.0 specification [5] defined the GET, POST and HEAD methods. The OPTIONS, PUT, DELETE, TRACE and CONNECT were added later in the HTTP/1.1 specification.

HTTP fixed most of the vulnerabilities that were present in FTP but, it still leaves room for some optimization. Some of the major drawbacks of HTTP when downloading a file over the internet include:

- Prolonged HTTP downloads results in wastage of megabytes in the form of lost or redundant bytes.
- The maximum download speed that a user can achieve can be easily limited / shaped by the web servers

B. BitTorrent

The major protocol that uses concepts of segmented file transfer is BitTorrent. BitTorrent is a communications protocol of peer-to-peer file sharing ("P2P") which is used to distribute data and electronic files over the Internet. The protocol was designed by Programmer Bram Cohen, a former University at Buffalo student in April 2001 and he released the first available version on 2 July 2001, the most recent release being in 2013. As of 2013, BitTorrent is has 15 - 27 million concurrent users at any given point in time [6] and it was also responsible for 3.25\% of all worldwide bandwidth, which is more than half of the total 6\% of the bandwidth dedicated to file sharing [7].

Even though, BitTorrent has a large user base some of the major disadvantages include:

- Files can be downloaded using BitTorrent only if sufficient seeders are available to "seed" the file
- Even though BitTorrent is a collaborative distributed platform for content sharing, many users do not "seed" the files that they have downloaded, thereby making it difficult for other users to download the file.
- A study [8] claims that 18% of all executable programs available for download contained malware.

- Another study [9] claims that as much as 14.5% of BitTorrent downloads contain zero-day malware, and that the protocol was used for distribution of 47% of all zero-day malware that has been found.

It is now safe to conclude that the existing technology is either extremely vulnerable to attacks or it wastes a lot data and thus resulting in wasted data in the user side or it doesn't allow a user to use his/her internet connection to its full potential.

III. BYTE SERVING

Byte serving is the process of sending only a portion of an HTTP/1.1 message from a server to a client.

A. Working Principle

Byte serving begins when an HTTP server advertises its willingness to serve partial requests using the Accept-Ranges response header. A client then requests a specific part of a file from the server using the Range request header. If the range is valid, the server sends it to the client with a 206 Partial Content status code and a Content-Range header listing the range sent. If the range is invalid, the server responds with a 416 Requested Range Not Satisfiable status code.

B. Applications

In the HTTP/1.0 standard, clients were only able to request an entire document. By allowing byte-serving, clients may choose to request any portion of the resource. Thus, byte serving can be considered as a method of bandwidth optimization.

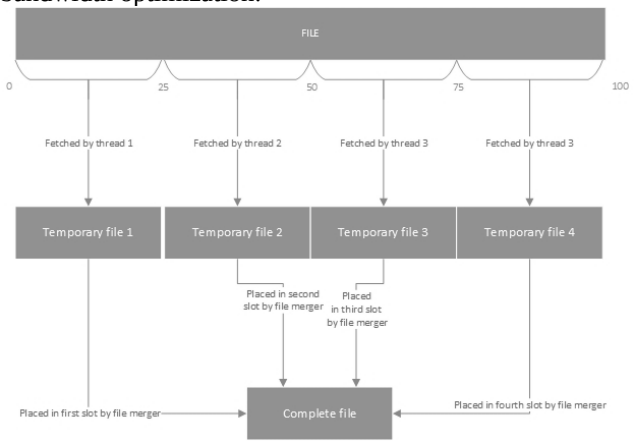

Fig. 1 Byte serving aids segmented file transfer

Byte serving is essential when a server is serving video files, if the server lacks this feature, then the video file may not be playable until the complete video file has been downloaded by the client, and seeking may be disabled. This technology can also be used to request for specific pages from a document in PDF from a server that is serving documents properly formatted in accordance with PDF.

Byte serving can also be used by multihomed clients to simultaneously download a resource over multiple network interfaces. To achieve this, multiple HTTP sessions are established and different segments of the same file is downloaded via the available network interfaces and the segments are then reassembled at the client. (refer figure 1)

IV. IMPLEMENTATION

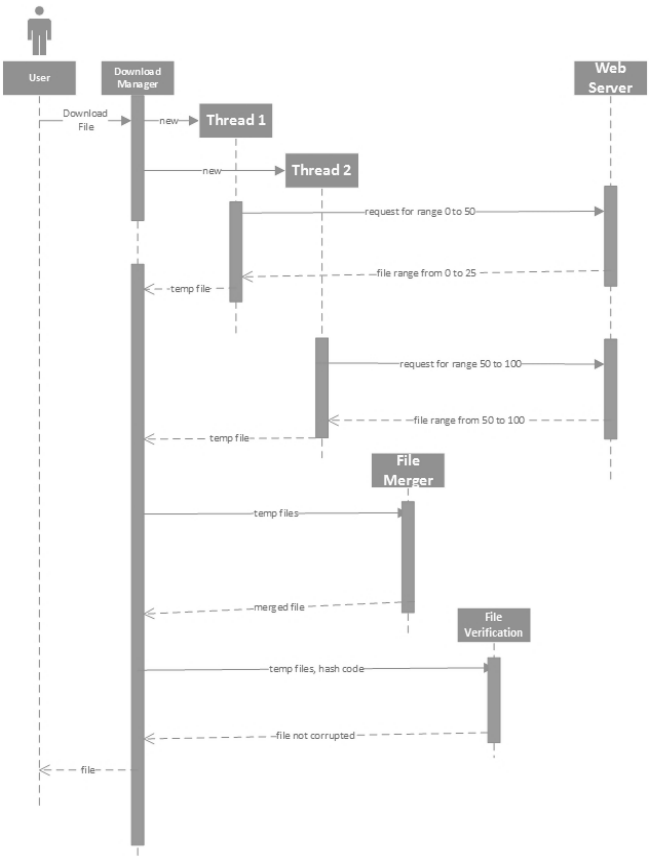

Fig. 2 UML sequence diagram for successful transmission with two segments

In order to implement segmented file transfer, the following steps must be carried out.

A. Number of Segments

Segment here indicates a portion of a file. Number of segments here indicates, the number of parts into which the file must divided into.

There are two ways of doing this, namely*:*

- All the files are split into the same number of segments
- The number of segments are decided dynamically, that is, the number of segments are decided based on speed of the connection, size of file etc.

Once the number of segments have been chosen, as many threads as the number of segments must be created. Each thread will download the segment of the file that has been allocated to it.

B. Data Structure

The data structure that is mainly required is a buffer. One buffer is required for every segment of the file that is being downloaded simultaneously.

A buffer is required to store the data that has been received from the sever. Once the buffer is full, the contents of the buffer are written to a temporary file that is stored on the disk. Once the download is complete, the contents of all the temporary files are merged.

Some of the types of software buffers that can be used include:

1) *Linear Buffer*: A liner buffer is defined as a region of physical memory that is used to store data temporarily. In this, the buffer is used to the data that is received from the server. The buffer is implemented using first in first out (FIFO) strategy. One of the major disadvantage of this type of buffer is that the contents of the buffer cannot be cleared, until the pointer to the front of the buffer passes the pointer pointing to the rear of the buffer.
2) *Concurrent Circular Buffer*: A circular buffer, circular queue, cyclic buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams. This buffer has a first in first out data characteristic. Ring buffers are quite common and are found in many embedded systems. A circular buffer has two indices to the elements within the buffer. The distance between the indices can range from zero (0) to the total number of elements within the buffer. The use of the dual indices means the queue length can shrink to zero, to the total number of elements. One of the major advantages of this type of a buffer is that all operations on the buffer is of constant time, i.e., no looping constructs are required to access or consume the elements in the buffer. The other benefit of a circular buffer is, that you don't need infinite amounts of memory, since older entries get overridden automatically. A concurrent ring buffer is a variation on the basic ring buffer. In this buffer, the two different indices that are user are updated by different threads. This means that one thread will continuously write into the buffer by taking control of the input stream and the front index. The other thread will take control of the output stream and the read buffer. This ensures maximum efficiency in buffering the data. The processor will be continuously writing as well as reading and there will be no need for the processor to wait for either of the operations to finish.

Once the concurrent data structure for each thread's buffer has been decided upon, the synchronization mechanism must be decided.

C. Synchronization Mechanism

When segmented file transfer is implemented, each segment of the file is downloaded on a separate thread and moreover, each thread will have a concurrent buffer, that is two threads will be continuously updating the buffer. One thread will be filling the buffer (adding the data that has been received from the server to the buffer) and the other will be emptying the buffer (this is, writing the contents of the buffer on to the disk).

In the first case, (each thread downloading a separate segment of the file) no explicit synchronization mechanism is required as each thread will establish a separate connection to the server (the connection that one thread establishes, will be independent of the other threads) and each thread will empty its buffer on to a separate file (the file that one thread writes into will be different from the other thread).

Synchronization will be mainly required at the data structure level, as two threads needs access to a common data structure, that is, the software buffer. The major reasons why synchronization is required are:

- To ensure that the rear index does not go ahead of the front index
- To ensure that the same data is not read from the buffer more than once
- To ensure that data is not overwritten before it is read exactly once

This problem is exactly the same as the bounded buffer problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

This problem can be solved using semaphores as illustrated by the pseudo code below:

```
mutex buffer_mutex;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer ()
{
   while (true)
  {
      item = produceItem ();
      wait(emptyCount);
      wait(buffer_mutex);
      putItemIntoBuffer(item);
      signal(buffer_mutex);
      signal(fillCount);
   }
}

procedure consumer ()
{
   while (true)
   {
      wait(fillCount);
      wait(buffer_mutex);
      item = removeItemFromBuffer ();
      signal(buffer_mutex);
      signal(emptyCount);
      consumeItem(item);
   }
}
```

Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock. It is important to note here that though mutex seems to work as a semaphore with value of 1 (binary semaphore), but there is difference in the fact that

mutex has ownership concept. Ownership means that mutex can only be "incremented" back (set to 1) by the same process that "decremented" it (set to 0), and all others tasks wait until mutex is available for decrement (effectively meaning that resource is available), which ensures mutual exclusivity and avoids deadlock. Thus, using mutexes inadequately can stall many processes when exclusive access is not required, but mutex is used instead of semaphore.

This problem can also be solved using monitors as illustrated by the pseudo code below:

```
monitor ProducerConsumer
{
   int itemCount;
   condition full;
   condition empty;

   procedure add(item)
   {
      while (itemCount == BUFFER_SIZE)
      {
         wait(full);
      }
      putItemIntoBuffer(item);
      itemCount = itemCount + 1;
      if (itemCount == 1)
      {
         notify(empty);
      }
   }

   procedure remove ()
   {
      while (itemCount == 0)
      {
         wait(empty);
      }
      item = removeItemFromBuffer ();
      itemCount = itemCount - 1;
      if (itemCount == BUFFER_SIZE - 1)
      {
         notify(full);
      }
      return item;
   }
}


procedure producer ()
{
   while (true)
   {
      item = produceItem ();
      ProducerConsumer.add(item);
   }
}

procedure consumer ()
{
   while (true)
   {
      item = ProducerConsumer.remove();
      consumeItem(item);
   }
}
```

Since mutual exclusion is implicit with monitors, no extra effort is necessary to protect the critical section. It is also noteworthy that using monitors makes race conditions much less likely than when using semaphores. Note the use of while statements in the above code, both when testing if the buffer is full or empty. With multiple consumers, there is a race condition where one consumer gets notified that an item has been put into the buffer but another consumer is already waiting on the monitor so removes it from the buffer instead. If the while was instead an if, too many items might be put into the buffer or a remove might be attempted on an empty buffer.

Once the threads have been properly synchronized, the next step is to decide on method to verify data integrity.

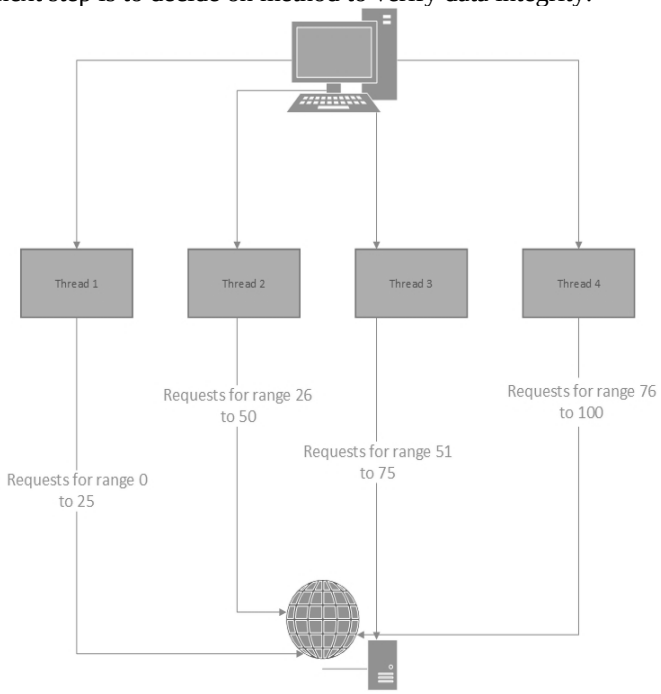

Fig. 3 Threads

D. Verify Data Integrity

Once the various temporary files that were created in the process of downloading have been merged, the next major aspect is to verify data integrity by using the checksum or hash that the server provided.

Usually MD5, SHA-1 or CRC-64-ECMA hashes are preferred due to the ease of implementation and fast computation on most systems that do not have much computation power. But MD5 and SHA-1 are considered to be cryptographically weak with respect to protecting data integrity and hence layered hashes and checksums like WHIRLPOOL, SHA-256, SHA-512 and CRC-64-ECMA must be preferred.

V. ADVANTAGES

Some of the advantages of Segmented File Transfer include the following:

- It saves some transmission capacity, as the number of lost or redundant megabytes is minimal compared to losing a prolonged HTTP or FTP download.
- Large files can be made available efficiently to many other users by someone who does not have large upload bandwidth. (Segmented Upload)
- Routes to the more obscure parts of the Internet can assert themselves across most of the Internet — this is especially true for dial-up users. (Segmented file transfer over a peer to peer network)

VI. LIMITATIONS

One of the major limitations of segmented file transfer is data integrity. Most naive implementations of segmented file transfer result in varying levels of file corruption.

A. Overcoming the limitations

Most software that implement segmented file transfer use a checksum or a hashing function to verify the data integrity. Layered hashes like SHA-256, SHA-512, CRC-64-ECMA (for individual segments) are used to guarantee data integrity.

VII. RESULTS

A software that implemented was made. The software always used 4 segments to download files and the buffer that was chosen was a concurrent circular buffer. Monitors were used to ensure proper synchronization.

Files of various sizes ranging from 250 MB to 1 GB were chosen and were downloaded multiple times and the average time taken to download the file was calculated accurate to the nearest second. The internet connection had a constant bandwidth of 50 Mbps. The test PC was connected via Ethernet and no other software that access the internet was running and the files were downloaded. The results obtained are as follows.

TABLE I
DOWNLOAD TIME USING SERVER I

File Size (GB)	Without Segmented File Transfer (s)	With Segmented File Transfer (s)
0.25	47	42
0.50	95	85
0.75	143	100
1.00	190	150

TABLE II
DOWNLOAD TIME USING SERVER II

File Size (GB)	Without Segmented File Transfer (s)	With Segmented File Transfer (s)
0.25	70	45
0.50	122	89
0.75	195	109
1.00	223	157

From the two tables and the two graphs, we can infer that the time taken to download the files from server 1 with and without segmented file transfer are more or less the same mostly because the server was idle and hence it did not matter if the file was downloaded at one stretch or using multiple segments.

But, there is a lot of variation in the download time for server 2, this is because the sever was likely loaded and sever would have opted to cap the maximum bandwidth that was allocated to a particular session and hence using multiple sessions decreased the download time by a huge extent.

Server 2 is considered to be a more realistic scenario. Hence, we can conclude that by using the concept of Segmented File Transfer, the download time can be reduced by a huge extent.

VIII. CONCLUSION

This paper presented a method of implemented an application that uses the concept of segment file transfer to overcome most of the disadvantages of the currently used technology.

This concept can also be extended such that if multiple users are connected to the same network and if one or more of them require the file, then each host can download a portion of the file that has been allocated to it and once the download has been completed, all the partial files can be sent to a common host for merging. Once the file has been merged, it can be sent to all the other nodes in the network that need have requested for the file.

The major advantage of using the concept of segmented file transfer (both upload and download) files over the internet is that the user will be able to utilize his or her internet connection to the maximum as the amount of data that is wasted will be reduced as well as the amount of time needed to transfer the file will also reduce without compromising on the security of the data.

REFERENCES

[1] Forouzan, B.A. TCP/IP: Protocol Suite (1st ed.), New Delhi, India: Tata McGraw-Hill Publishing Company Limited, 2000.

[2] nurdletech. Securing FTP using SSH.

[3] Kozierok, Charles M. The TCP/IP Guide v3.0, 2005.

[4] Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Nielsen, Henrik Frystyk; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim. Hypertext Transfer Protocol -- HTTP/1.1. IETF. RFC 2616, June 1999.

[5] Berners-Lee, Tim; Fielding, Roy T.; Nielsen, Henrik Frystyk. "Method Definitions". Hypertext Transfer Protocol -- HTTP/1.0.

[6] Wang, Liang and Kangasharju, J. Measuring large-scale distributed systems: Case of Bit Torrent Mainline DHT, 7 January 2016.

[7] Palo Alto Networks. Application Usage & Threat Report, 7 April 2013.

[8] Berns, Andrew D.; Jung, Eunjin (EJ). Searching for Malware in Bit Torrent, 24 April 2008.

[9] Vegge, Havard; Halvorsen, Finn Michael; Nergård, Rune Walsø. Where Only Fools Dare to Tread: An Empirical Study on the Prevalence of Zero-Day Malware, 2009.