# BitTorrent with Congestion Control

HingOn Miu (hmiu@andrew.cmu.edu) Xing Zhou (xingz@andrew.cmu.edu)
Carnegie Mellon University
*24–10–2013*

## Introduction

The peer program mimics a peer user in the BitTorrent network. Each peer has partial or full data of a complete file, along with the hashes of the file. The peers can then communicate through UDP to get the missing file chunks to complete the file.

## Peer-to-Peer

Upon start up, each peer is given a list of available peers, a list of file chunks the peer currently owns, and the master chunk file list. The peer loads the hashes it owns into memory and get ready to receive either incoming query requests or user inputs.

Each peer can be functionally separated into two mutually exclusive parts: a peer can be a client or a server. It is a client when it processes a user request and requests chunks from other peers. It is a server when it sends data in response to a query. The server side implements the sliding window algorithm for flow control, while the client side handles contacting servers for retransmits.

## Operation

The operation of each peer happens in stages named set up, discovery, data collection, and data construction.

### Set up stage

The set up stage is exactly what it sounds like. It begins as soon as the exec is called. In this stage, each peer parses its own haschunk file and loads its own chunk hashes into memory. This stage ends when all the chunk hashes have been loaded and the peer is ready to play the role of either client or server.

### Discovery stage

The discovery stage begins once the user has typed a GET request. The peer would parse the request and extract the requested chunk file. It'll load the hashes in the chunk file into memory. Then it'll issue WHOHAS requests with the list of hashes to all of its peers in its map file. The stage completes once the peer - client - receives at least one IHAVE claim for each missing chunks and it moves into the data collection stage.

### Data Collection stage

Data collection is when the client issues GET requests to the peers that contain the chunks desired. In response, the servers should send DATA packets containing the data chunks that the client will eventually assemble. It is in this stage that a fate-sharing "transfer" connection is established, and so this "transfer" connection manages the states of the transfer, such as the sliding window algorithm and retransmission.

During Data Collection, new IHAVE received from peers will also be listed as one of the available peers of the missing chunk. Hence, technically, during Data Collection, discoveries of missing chunks' ownership are still processed. Each client and server keeps a list of active transfers, and the data collection stage ends once the list is empty.

### Data Construction stage

Data construction stage is when the data from the peers are output to the file requested by the user.

## Reliable Data Transfer

The peer is implemented with TCP-like congestion control. The sender side congestion window size is increased or decreased during the Slow Start and Congestion Avoidance mode. Every timeout of ACK packets or duplicate ACK packets mark the transitioning between SS and CA mode.

Slow Start mode increases the window size per ACK packets received, so the increase should be exponential. Congestion Avoidance mode increases the window size per round-trip time, so the increase should be linear. The only decrease in window size would happens between the transition from CA to SS mode when a loss of packet is detected, the window size is then reset to 1.

## Data Structures

There are three main data structures used in this project. The most crucial one is the linked list data structure, which is defined in linkedlist.h/c. This is the generic single-linked linked list data structure with appropriate manipulation functions. The data it stores is agnostic and thus is used throughout the code to main any dynamic lists.

The second structure is the chunk structure, defined in chunk.h/c. This chunk structure associates the chunk id, hex hash, and binary hash. All chunks are created in the setup stage and populated as chunks are received.

## Testing Methodology

Peers Setup Configurations:
-- 2 peers, mutually exclusive ownership of the files
-- 3 peers, mutually exclusive ownership of the files
-- 3 peers, duplicate ownership of the files among 2 peers

1.) We run multiple peers in the network (p1, p2, p3,...), and we tested p1 with one user GET request that fetch data from all other peers p2, p3, ...

2.) During the data transfer we shutdown peers and we let the data fetching peers begin new fetches with other available peers instead.

3.) We tested with multiple simultaneous user GET requests (each p1, p2, p3,... are getting and sending data packets to each other).

4.) Multiple sequential user GET request for one peer. After a peer is done and prints "GOT ...", we enter more GET requests to the peer for identical or different data.

5.) Tests are repeated with and without congestion simulation.

## Vulnerabilities

For now, the local peer pre-loads all chunk data of the chunks listed in the has_chunk_file. Hence, it is a potential burden on running time and memory allocation if the local peer's has_chunk_file contains too many hashes.

Since each chunk is 512 * 1024 = 524288 bytes and that different machines have different runtime memory allocation size, there is a risk of running out memory to allocate buffers to store the chunk data in the peer setup stage. Because malloc or calloc returns NULL if it runs out of memory to allocate, we have properly handled those cases to avoid segmentation fault.

Our implementation is that the peer floods the network with WHOHAS packets until each missing chunk has at least one IHAVE reply before the peer sends any GET packets (sort of like waiting for all missing chunks to be claimed before fetching data). For example if c1, c2, and c3 are missing, and if the network does not have c3, the peer does not send GET for c1 and c2, and waits for c3 to be claimed before sending GET for c1, c2 and c3.

Hence, while this strategy avoids fetching data that might end up undeliverable to the user (if c3 never shows up in the network), it creates inefficiency since the peer waits for all missing chunks to be ready to be fetched instead of getting whatever is available now and getting those that with delayed IHAVE replies later.