File Distribution Protocol

# Analysis: Multicast vs Peer to Peer

Multicast is ideal for sending the same message to multiple nodes. The transmission time will be constant for any number of nodes. Until wider IPv6 support, with its mandatory multicast, Multicast traffic is confined to local subnets. A Multicast host transmitting at a constant speed is making the assumption that all of its listeners can download at that same speed. If a client cannot consume the data at that rate, it will drop a substantial amount of packets. This could be avoided by transmitting at a variety of speeds, however this is potentially a complex solution which could impact on performance.

A peer-to-peer system in theory should scale well as the number of nodes increases, though not as well as Multicast. P2P systems can be extremely complex to develop, debug and maintain, and this is a considerable disadvantage of P2P. However using a small amount of centralised infrastructure they can be simplified and made easier to manage. A central BitTorrent-style tracker could be used to avoid requiring something like a Distributed Hash Table. P2P traffic is accepted on the IPv4 internet. P2P should also handle peers dropping in and out. P2P over TCP will not suffer from the slow consumer problem as TCP’s congestion control features will kick in. P2P will also be able to scale better for retransmitting chunks (e.g. if packets are lost or a host goes down for a short period of time). A multicast system must either loop through data continuously (teletext-style), or receiving peers would ask the host to re-send chunks. Resending chunks will waste the time of every client which already has that chunk.

## Performance Predictions

To analyse the two methods further I decided to calculate the expected performance of Multicast and Peer-to-Peer. The times used in these calculations are based on a 1GB file being transferred over a 100Mbit/s connection. In these calculations it is assumed that each node can upload and download 100Mbit/s at the same time.

Figure 1.0 – Graph predicting file transfer time for different architectures on a lossless network. Note: Lines for Multicast and P2P (Chunked) largely overlap.

Figure 1.0 shows that an un-chunked P2P system will be considerably out-performed by Multicast, but will still be significantly more efficient than Serial transfer. In these calculations, Chunked P2P comes across as performing nearly as well as Multicast. I think these calculations assume the switch backplane capacity is enough to cope with every node being maxed out.

## Empirical Research

Before I select which architecture to use in my file distribution system I decided some tests were needed to verify my predictions and assumptions, particularly after the surprising result of Chunked P2P in Figure 1.0.

### Test 1: Client-Server Serial transfer

Basic file distribution system used as a benchmark. A single host machine, running iperf will distribute the required amount data. UDP mode used to avoid TCP ramp up artefacts and any packet corruptions.

### Test 2: Multicast transfer

Using iperf again, this test will measure the time it takes for all clients listening to the group to receive all of the data.

### Test 3: BitTorrent Peer to Peer

Using the most complex software of the 4 tests, this experiment should test whether the performance of a pre-existing P2P file distribution solution matches up with my expectations (Somewhere between Unchunked and chunked P2P in Figure 1.0). I will be using Herd (Garrett & Gadea, 2014) as a P2P system for this test. Herd uses BitTorrent underneath to perform the file transfers.

## Results

Figure 1.1 – Empirical data of Serial, Multicast and a P2P implementation transferring a 1GB file across a subnet to a variable number of nodes. In grey are the values from Figure 1.0

The results stick very closely to the predicted values. The multicast test performs better than P2P, with the transfer taking exactly the same amount of time for any of the numbers of nodes recorded. The P2P software tested performs better than I expected. It does not outperform the predicted “P2P Chunked” however as noted above that value was optimistic.

## Analysis Conclusion

Although it was outperformed in the Empirical Data, P2P seems to be the most appropriate architecture to use for this tool. By choosing P2P the application will support networks outside Multicast-enabled subnets. The speed advantage is not great enough to warrant the restriction of this software for inside a subnet. The P2P architecture may require more careful designing, but the reliability checks will scale far better than for any pure-Multicast solution.

An ideal solution might be a combination of Multicast and P2P to provide the speed boost Multicast provides when used on a low packet-loss LAN subnet, using P2P for re-sending missing chunks.

# Design: Peer to Peer System

The distribution system must be reliable and efficient. My analysis above shows the way to bring efficiency is to go Peer to peer, distributing larger files in smaller chunks. This design section coverts the design of the Protocols, file structures, and system architecture. Security, Reliability, Maintainability and Efficiency were the main goals (roughly in that order) in mind when designing this system.

## Initial Metadata

To be able to join a swarm, and obtain chunks a peer must be in possession of the metadata of the swarm. This metadata will be stored in a JSON-structured file, a file extension of .p2pmeta is suggested.

|  |  |
| --- | --- |
| Swarm Metadata | Description |
| Hash Type | Identifies the checksum algorithm. Important for future security, SHA-256 may eventually not be secure enough. |
| Metadata Hash | Acts as an ID for the Swarm, ensures that an innocent peer won’t join a swarm with a corrupt metadata file. This digest protects the HashType, SM Hostname, filenames, chunk hashes and file digests for each file/chunk. |
| Swarm Manager Hostname | A peer joining the swarm will register with this Swarm Manager. |
| List of file metadata: |  |
| Filename | Used to correctly name the downloaded file |
| File Hash | Ensure the entire file is transferred and pieced together correctly |
| List of chunk metadata: |  |
| Chunk size | Allows file size to be calculated |
| Chunk hash | Allows integrity of chunk data received from peer to be verified. |

## Distribution Architecture

The distribution system will contain two main applications: The Peer and the Swarm Manager. A smaller third application will create the initial metadata file based on input files.

### Swarm Manager

The Swarm Manager will maintain a list of active peers which have registered for the swarm. Peers will send a register command regularly to stay on the list of peers for a particular swarm. Peers will then be able to request the list of active peers for a swarm. A swarm is uniquely identified by the Metadata hash in the initial metadata.

### Peer

When started, each peer will use the information in the .p2pmeta file to check for the existence of the files to be downloaded. If the files do not exist, the peer will create them and allocate enough hard drive space for the download. If the files exist the peer will verify the integrity of the files and chunks to determine which file(s) or chunk(s) are missing. This architecture should support pausing/resuming of downloads without any issues and without requiring any special shutdown code to run (supporting sudden power off).

#### Peer Behaviour

Peers looking to acquire chunks in the swarm will connect to the Swarm Manager using the swarm metadata. The peer will then connect to other peers using information obtained from the Swarm Manager.

When a peer connects to another peer, they will exchange a list of chunks they each possess. The pair can then request appropriate chunks from each other simultaneously. When both of these peers are complete, they will disconnect from each other.

The algorithms for peer selection and chunk selection will be important to ensure efficient use of resources in the swarm. A very basic algorithm will be used first (choose the lowest chunk ID/first peer in list), but if time allows this should be improved later.

## Network Protocol Design

### Swarm Manager Protocol

JSON is used to encode the messages sent to/from the Swarm Manager. JSON is a good candidate for this as it is a widely implemented data exchange format. All data exchanged at this step is Human readable, allowing for easy encoding and handling. The human readability should ensure it is easy to implement, and easy to debug the Swarm Manager. JSON if handled correctly is also very compatible with future changes, as old clients should just ignore new fields in the JSON object.

See “Swarm Manager Messages.json” for example messages the Swarm Manager handles/sends.

### Peer to peer protocol

Unlike the Swarm Manager protocol, peers will exchange binary data. I considered two ideas for this protocol: firstly an FTP-like system where JSON would be exchanged on a “control” socket, and a separate socket would be set up for the sending of binary data. The second idea was to have a small header which split up control vs data messages on the same socket.

Operating multiple sockets between peers would make the protocol simpler, but it would make the program harder to allow through firewalls (Multiple ports would have to be opened). Using a single socket would also reduce the latency between agreeing to exchange chunk data and actually sending/receiving chunk data as TCP sockets have a 3-way handshake which could slow things down.

#### Peer to Peer Message format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  |  | Bits | |  |  |  |  |  |  | 15 |
| Version | | | | Msg Type | | | | Message length | | | | | | | |
| Message length | | | | | | | | | | | | | | | |
| Data | | | | | | | | | | | | | | | |
|
|
|

Figure 2.0 – P2P Message Header Format.

In this implementation, the version is 0000, and the message type is either 0001(Data) or 0000(Control).

For control messages, Data should be interpreted as a JSON object. See “Peer Messages.json” for examples.

For Data messages, the message data should be interpreted as follows.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  |  |  |  |  |  | Bits | |  |  |  |  |  |  | 15 |
| Hash Length | | | | | | | | Metadata Hash | | | | | | | |
| Metadata Hash (Variable length)… | | | | | | | | | | | | | | | |
|
|
| File ID | | | | | | | | | | | | | | | |
|
| Chunk ID | | | | | | | | | | | | | | | |
|
| Chunk Data….. | | | | | | | | | | | | | | | |
|
|
|

Figure 2.1 – P2P Data Message Sub-Header.

# Implementation

## Testing

Two main testing strategies were used: Unit testing and some more general ‘white-box’ testing.

Unit testing in this project is mainly focused on areas in which subtle bugs would cause potentially hard to track down problems: Message Parsing, File Parsing and Message serialisation. Some of the Swarm Manager general logic has good test coverage as well as I tested out Mockito, a mocking-based unit testing framework. I found it to be slightly less than convenient though, so the Peer code has less good test coverage.

More general testing was performed by testing specific features (Such as requesting chunks from multiple peers, or resuming a partial download) at a time. By attempting to minimise other variants it helped make debugging just about possible. In general I found debugging P2P networking code very challenging due to the high numbers of connections, threads and possible points of failure. One strategy for preventing hard-to-debug problems, which I worked out quite late into development, was to treat one peer-to-peer connection as very disposable, shutting down the socket as soon as anything even maybe went wrong (Short timeouts, etc). This behaviour reduced problems such as chunks being stuck “INPROGRESS” and minimised issues relating to the java program not shutting down because a thread wouldn’t exit correctly and then would block on join().

## Issues Encountered

### Chunk & Peer selection algorithms

During an early pass of the peer code the Chunk selection and peer selection algorithms were drafted as roughly: “Choose an available Chunk/Peer with the lowest ID number”. This “algorithm” made it all the way through the main development process and it wasn’t until fairly extensive testing (10-15 nodes transferring 250MB+ files) that the problems of this method really began to show.

The most obvious side effect of these basic algorithms was in peer selection. If the peer at the top of the peer list (as retrieved from the swarm manager) disconnected, then a peer still left in the swarm is stuck attempting to connect to this peer until the Swarm Manager finally prunes the disconnected peer from the list. This issue could also have been avoided by using an effective choking system. My choking implementation was ineffective because the results of choking a peer were overwritten by the frequent peer list downloads from the swarm manager.

Discuss Chunk + Peer selection algorithm importance.

# Extras

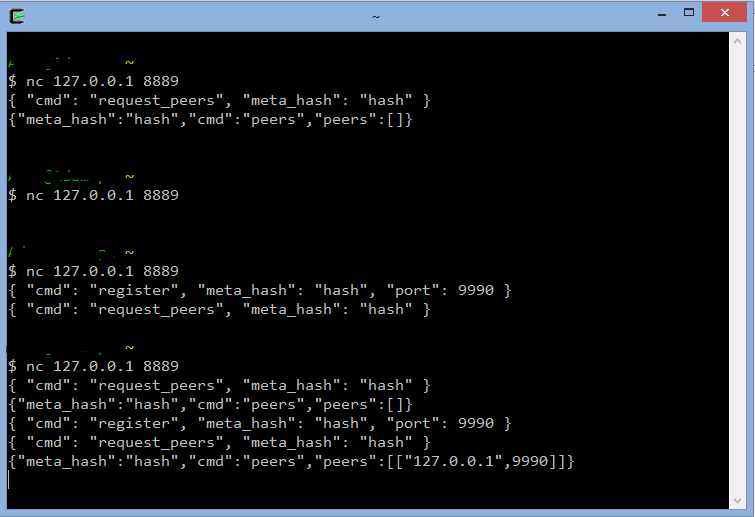
Compression? File content / file extension testing.

Discuss compression – best served outside of this P2P system. A layer on top could do this, a system like Herd’s one liner would be nice.

Security – Prevent malicious peers / malicious network. Trust host (or “.torrent file”).

* Compression

# Testing



# Evaluation

## Functionality

## Code Quality

## Improvements

* De-couple connecting to a new peer and getting the peer list. Peer list updates should be in another thread.
* JSON messages get very large towards end of large file transfer (huge chunk list). Compressing this message, or only sending a differential update would be far more efficient.
* The current chunk must finish downloading, and then the receiving peer must request the next chunk before the sending peer starts sending the next chunk. This introduces some delay which may cause significant slowdown on a high latency network.
* Toilet paper output.

# Appendix

## Graph Figures – Data and Methods

### Figure 1.0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nodes (**n)** | Serial | Multicast | P2P (Unchunked) | P2P (Chunked) |
| 1 | 85.89934592 | 85.89934592 | 85.89934592 | 85.89934592 |
| 5 | 429.4967296 | 85.89934592 | 196.1368399 | 85.983232 |
| 10 | 858.9934592 | 85.89934592 | 251.5964572 | 86.0880896 |
| 15 | 1288.490189 | 85.89934592 | 285.0337001 | 86.1929472 |
| 50 | 4294.967296 | 85.89934592 | 386.4787957 | 86.9269504 |

|  |  |  |  |
| --- | --- | --- | --- |
| Chunk Size(**C**) | Total File Size (**D**) | Bandwidth (**B**) |  |
| 262144 | 1073741824 | 12500000 |  |

|  |  |  |
| --- | --- | --- |
| Architecture | Total transfer time formula | Explanation |
| Serial |  | Bit-containing packets served serially. Total bits to serve |
| Multicast |  | Same packets served to all clients simultaneously. |
| P2P (unchunked) |  | Upload begins only after download is complete. Each subsequent client downloads with 1 more seeder than the last. |
| P2P (chunked) |  | The optimal architecture for speed is a chain of peers, links long.  The total time is the time until the last peer has downloaded the last chunk.  This will be the time taken to transfer the last chunk to the first link + the length of the chain.  number of chunks |

## Example Output

### Example output CSV file from -stats

### Example Program Run through (With simulation)

# References

Garrett, R., & Gadea, L. (2014, September 25). Herd: A single-command bittorrent distribution system, based on Twitter's Murder. Github. Retrieved from https://github.com/russss/Herd