File Distribution Protocol

# Analysis: Multicast vs Peer to Peer

* Multicast requires only transmitting data once for many to receive.
* Multicast has extremely large limits.
* Multicast should have the best file transfer time on a perfect communication channel
* Multicast does not transit over public internet
* On a local subnet multicast should perform the best.
* Some form of peer to peer should scale relatively well. May take sometime to ramp up though.
* On a local subnet peer to peer may not make the best use of bandwidth due to Ethernet multicast etc.
* Peer to peer will work over the internet
* Peer to peer may be able to cope much better with the central host going down if other peers have the file.
* Peer management in P2P might be complex. Old style Bit Torrent used trackers, a centralised server. New Bit Torrent uses DHT’s and other types of local peer discovery. Very complicated though.
* Peer to peer can handle varying peer speeds. Multicast will transmit at one fixed speed. Slow receiver/consumer.
* Chunked / unchunked P2P. Chunked will reduce the lag time between a peer downloading data and then beginning to upload the same data to another node.

Peer to peer will 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). With multicast either the host could loop through data continuously, or receiving peers would ask the host to re-send chunks, but this may require the host to be connected to every receiving peer which isn’t so scalable. Also, when resending every peer will receive the chunk again, rather than only the peer which needs it.

## 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 visualises a number of important facts:

1. All architectures take the same amount of time if number of nodes (n) =1.
2. Serial transfers do not scale well as n increases.
3. P2P (unchunked) demonstrates a log(n) relationship.
4. Multicast’s performance is not related to the number of nodes.
5. The performance of P2P (chunked) is modelled to be nearly identical to Multicast’s.

The last point, chunked P2P performance nearly equalling multicast performance was surprising. However I it stems from the assumption that all nodes will be able to transmit and receive at 100Mbit/s, regardless of other traffic on the network. Unless an extremely high performance switch/router is managing the subnet this is an unlikely scenario.

Though I have no mathematical predictions, I suspect in reality chunked P2P would provide a distinct improvement on unchunked P2P, but would not perform as well as Multicast on an average subnet due to the higher total network resources required.

## 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.

### Test 1: Client-Server Serial transfer

The most basic file distribution system used as a benchmark for the other two methods. A single host machine will serve the files to connecting clients.

I will measure using iperf (UDP mode) to attempt to avoid packet corruptions or TCP ramp up artefacts impacting results.

### 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

The Swarm Manager is a fairly simple program which allows peers to be added to the swarm currently exchanging chunks of a specific p2pmeta file.

I chose to use JSON to encode the messages sent to/from the Swarm Manager. JSON is a good candidate for this as the data is all text, not binary data, and JSON is a widely implemented data exchange format. Another useful feature of JSON is that it’s very human-readable, allowing easy debugging.

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 would cause a slight slowdown.

#### Message format

* 4bits version (0000 in this protocol version)
* 4bits message type (0000 Control, 0001 Data, room for future changes)
* 3bytes message length (max size 224 bytes per message). Note: excludes header 4 bytes. Little endian
* Data of length “message length” follows

If the message type is Control, the message data should be interpreted as JSON. Example message formats are described in “Peer Messages.json”

If the message type is Data, the message data should be interpreted as follows.

* 1 byte hash length (N)
* N bytes meta hash
* 4 bytes file ID Little endian
* 4 bytes chunk ID Little endian
* Chunk data

# Implementation

2 Distinct elements when running: Swarm Manager and Peer.

Discuss Thread issues?

Discuss Chunk + Peer selection algorithm importance.

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.

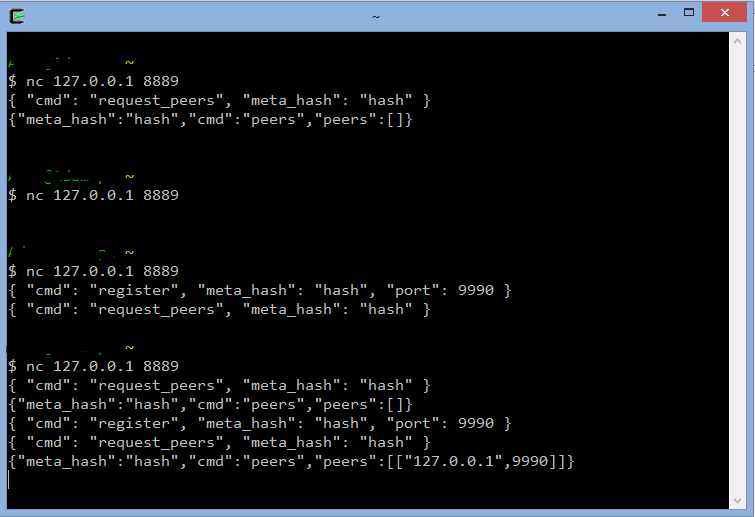
# Extras

Compression? File content / file extension testing.

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

* Compression

# Testing



# Issues Encountered

# Evaluation

## Functionality

## Code Quality

# Example Output

### Example output CSV file from -stats

### Example Program Run through (With simulation)

# 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 |

# 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