Name: Ramandeep Singh
Username: rs5150
Date: 22-12-2015
Section: Masters in Information Systems

# Assignment 10: Final Project

_____ Basic functionality [Max 10 points]
_____ Search for and reliably retrieve files [Max 30 points]
_____ Basic congestion control [Max 15 points]
_____ Support and Utilize Concurrent Transfers [Max 15 points]
_____ Congestion control corner cases [Max 15 points]
_____ Robustness: [Max 10 points]:
_____ Style [Max 5 points]
_____Total [Max 100 points]

Total in points ___________________
Professor's Comments:
Affirmation of my Independent Effort: _____________________________
(Sign here)

DESIGN SPECIFICATION

# BitTorrent Over UDP with Congestion Control

| | |
|---|---|
| Author: | Ramandeep Singh |
| Creation Date: | December 17, 2015 |
| Last Updated: | December 22, 2015 |
| Document Ref: | 1.0 |
| Version: | 1 |

**Approvals:**

Jean-Claude Franchitti ______________________________

# 1 DOCUMENT CONTROL

## 1.1 Change Record

| Date | Author | Version | Change Reference |
|---|---|---|---|
| 17-Dec-15 | Ramandeep Singh | 1 | |
| | | | |
| | | | |
| | | | |

# Contents

## 2 TECHNICAL OVERVIEW

This Design Specification documents the detailed design of BitTorrent like implementation over UDP protocol. The trick here is the use of UDP protocol which allows us to send light weight messages than TCP and handle congestion and flow control ourselves. There is no distributed tracker and every node has access to the central file which contains the list of nodes. The system however does support parallel downloads and uploads to different peers.

### 2.1 Challenges

Following is the list of challenges that are posed by implementing BitTorrent

- **Concurrency**: Handling parallel uploads and downloads can be challenging, as it requires ability to handle multiple requests, while ensuring data integrity is maintained. The implementation uses the concurrency packages under JAVA API, to ensure that code is highly concurrent and minimal synchronization overhead is there, as it uses **Atomic** constructs which are features present in modern platforms that ensure that high concurrency is maintained without synchronization overhead.

- **Reliability**: UDP is inherently unreliable, so reliability has to be implemented within the application. The application manages this by using sequence and acknowledgement numbers within the data payload of the UDP datagram. Also daemon threads are used to manage scenarios of data loss, packet reordering, lag, etc. In case the downloading peer finds that there is loss on the network or that 3 consecutive requests have failed with the peer, it stops downloading from the peer and re-broadcasts the request to download the chunk from another peer.

- **Congestion Control and Avoidance**: The application implements **slow start with Congestion avoidance** algorithm. This ensures that the window size increases by 1 for every successful ACK response up to a slow start threshold and if there is a packet loss the window size is lowered to its initial value. Also **FAST Retransmit** is implemented, so that if a node receives 3 duplicate acknowledgements, it will resend the data packet again without waiting for the timeout.

- **Handling Race condition:** As the requests are broadcast to all the peers in this implementation, there is the problem of handling race condition. The implementation handles this by ensuring that after the WhoHas request is issued a peer waits for a random amount of time before issuing GET request.

## 2.2 Building Blocks

### Building Blocks

- **Peer (Peer.java)**: This is the class that is designed to handle functionality for managing all the requests i.e. ACK, GET, DATA, WHOHAS, IHAVE and DENY. In this class in addition to ensuring reliability for DATA requests, I have implemented **reliability for GET** requests, handled timeouts for both uploading and downloading peer. This implementation ensures parallel downloads from multiple peers at the same time whilst ensuring restriction that a peer can only download or upload to a single peer, but may download and upload to other peers simultaneously. This class has a listener thread, which listens for incoming packet requests, this listener thread discards the packet, if the packets version and magic number do not match, 1 and 15441 respectively. The code in this class can handle **highly concurrent** requests without synchronization overhead as it uses Java's concurrency packages such as **AtomicLong**, **ConcurrentHashMap, CopyOnWriteArrayList,** etc**.**
- **ReliabilityAndCongestionHandler**: This is a nested class within Peer.java, this manages upload and download of data packets from a Peer. An instance is created for an Active peer upload and download, it manages reliability and congestion for upload and download of the data packets. It also removes un-responding peers whether they are uploading or downloading, this ensures that it can serve other requests and not block or wait indefinitely on bad peers. This functionality of removing bad peers is accomplished by using a daemon thread which checks whether an ACK or DATA packet has been received within the last 5 seconds and then handling the corresponding case accordingly.
- **Packet (Packet.java)**: This is just a java bean class that stores the packet information. As in java the unsigned data types are not supported, so the **packet header here is of 24 bytes**, similarly **chunk hash size is of 40 bytes**. Also as java does not support raw sockets, so the header information is encoded within the data portion of the payload. This leaves 1476 bytes for the data.
- **FileHelper (FileHelper.java):** This class is used for handling all file related portions including reading chunks of data from master data file, writing output **chunk and peer-problem** files to **temporary** directory.
- **SetupPeer (SetupPeer.java):** This is the entry point for the project, it handles user input, prints out help message for command usage on the command line, initiates a peer instance, and starts monitor thread of peer.

## 2.3 Class Relationship Diagram

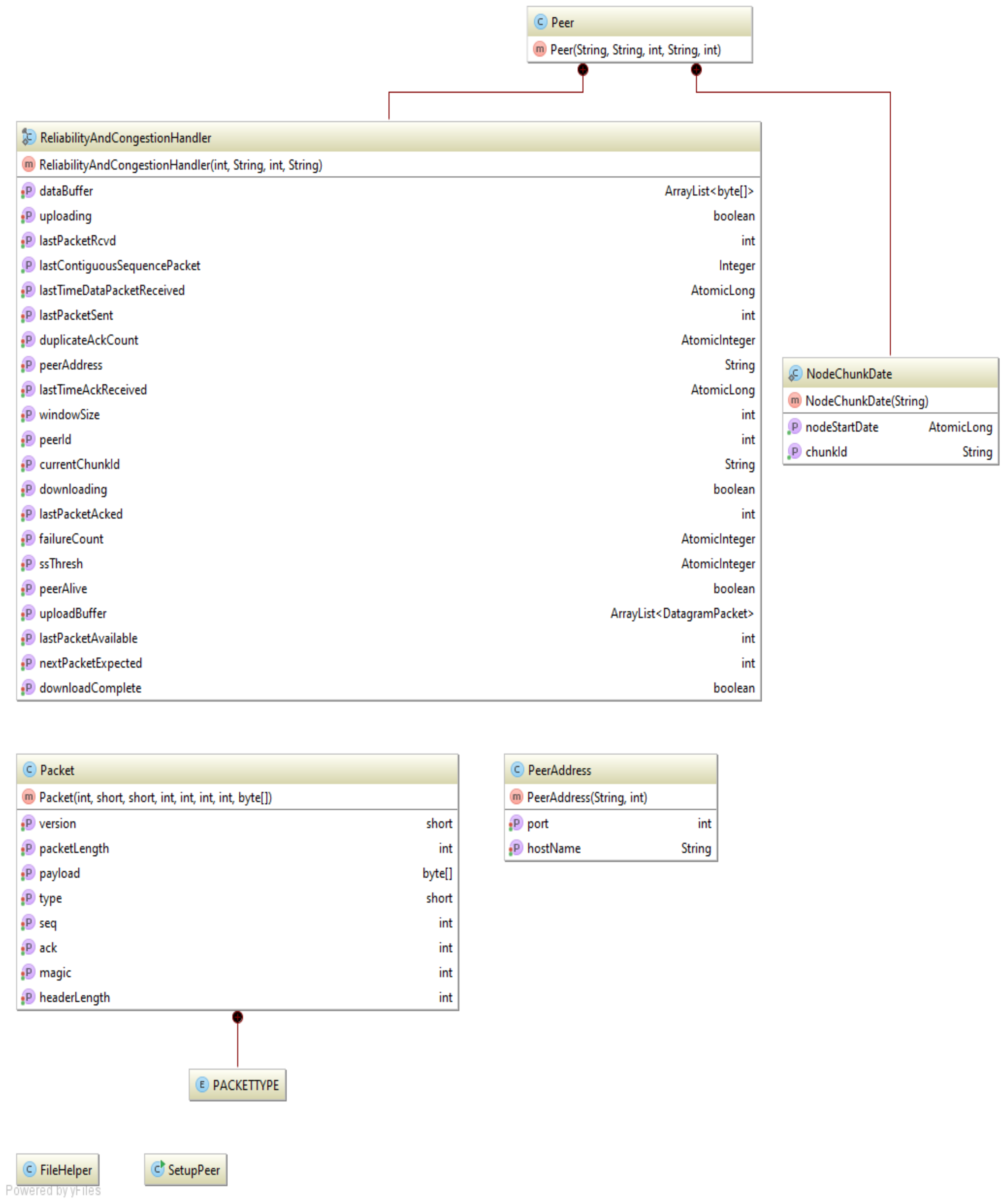


**Figure 1: Class Diagram Of implementation**

# 3 IMPLEMENTATION DETAILS

## 3.1 Header Structure

The packet structure used by the application is shown below. The size were changed from the required specification because java does not support unsigned data types. So extra bytes are needed to encode the same information. With **JAVA 8** integers can now be used to support unsigned data ranges, so we save on representing Sequence and Acknowledgement numbers. The total length used is 24 bytes.  The payload varies for different types of requests as mentioned in the requirement specification.

| | |
|---|---|
| MAGIC NUMBER (4 bytes) | |
| VERSION(2 bytes) | TYPE(2 bytes) |
| HEADER LENGTH (4 bytes) | |
| PACKET LENGTH(4 bytes) | |
| SEQ (4 bytes) | |
| ACK (4 bytes) | |
| PAYLOAD(1476 bytes or less depends on packet type) | |

## 3.2 Searching Peers

WHOHAS request is issued to all the peers in the peer list (excluding oneself). The requests are chunked to ensure that they fit into one packet. The main thread then sleeps and loops to ensure that all responses are gathered before choosing the nodes that it can send GET requests to in parallel. The wait here is random, to avoid race conditions.

## 3.3 Downloading chunks

The GET requests are issued in parallel to the nodes that have the chunks, ensuring that at one point of time only one chunk is being downloaded from a single node. GET Requests are also reliable in the sense, that if a request is lost or the node does not respond, WHOHAS request will be reissued. After the connection with a peer is established the chunk will be downloaded from the same peer, unless the node fails, in which case the WHOHAS request is reissued. After all the individual packets have been received successfully the SHA-1 hash is computed over the entire packets and compared with the CHUNK HASH ID, if they don't match the request is re-issued and the download process begins again. And after all the chunks are downloaded, the output file is created and chunks are written in order as specified in the GET request.

## 3.4 Uploading chunks

If a node receives WHOHAS request, it returns the IHAVE response for the chunks that it owns, After the GET request is retrieved it reads the chunks from the master data file and further chunks them into DATA Packets, It assigns a sequence number to the packet and then sends the DATA packet as response, On subsequent times it expects an ACK packet, on receipt of which it sends the next DATA packet in sequence until, the entire chunk has been transmitted.

## 3.5 Handling Concurrency

As alluded to earlier, handling concurrent requests can be challenging. As it involves various consideration such as concurrent data modification, race conditions, synchronization overhead. The implementation handles this by using highly concurrent constructs such as AtomicLong, ConcurrentHashMap, and CopyOnWriteArrayList, etc. In addition requests for download of chunks are distributed to different nodes at the same time. So performance is maximized while ensuring that the congestion is minimal. In this implementation first WHOHAS requests are issued till a node is able to find details of the nodes that have those chunks, then GET request is issued to each of those nodes in parallel, ensuring that each peer node is only asked for one chunk at a time. As GET requests are issued after waiting for a random duration of time, this implementation also avoids RACE condition.

## 3.6 Handling Reliability

Reliability is ensured by using sequence and acknowledgement numbers and use of timeouts. GET Request is also reliable in this implementation as if a peer finds out that the node to which it sent the GET request has not completed processing it within 60 seconds, the WHOHAS request is issued and the download process starts again.

For reliability of DATA and ACK requests a monitoring thread is used. The various scenarios are mentioned below.

- **Lost ACK**: A node waits for 5 seconds and if it finds out that no new DATA packet has been received, it assumes that the ACK packet was lost, so it retransmits it. This is done for 3 consecutive times after which the peer is marked as bad and the WHOHAS request is reissued.

- **Lost DATA**: A node waits for 5 seconds to receive an ACK, if it finds out that the node has not acknowledged the last data packet that it has sent, it retransmits the packet.

- **Out of order packet handling**: In this implementation, the receiving node discards the packets that are out of order and resends the ACK for the packet it expects.

- **Duplicate ACK**: The node discards duplicate ACK packets, but keeps track of this count, if this count is 3, it assumes that the data packet was lost and resends it.

## 3.7 Handling Congestion and Flow Control

This application implements the slow start algorithm with congestion avoidance and fast retransmit. Each of these implementation details are explained below.

- **Slow Start Algorithm**: After each successful ACK response the window size is increased by one till it reaches the slow start threshold, after which it stops increasing the window size. If a packet is lost however, the congestion avoidance algorithm comes into play.

- **Congestion Avoidance**: If an ACK is lost, the algorithm first reduces the slow start threshold to max (current window size/2, 2). And then reduces the window size to 1 and restarts the slow start approach.

- **FAST Retransmit**: If 3 duplicate acknowledgements are received by the sender, it assumes that the packet was lost and it resends the packet rather than waiting for the send timeout.

# 4 QUALITY OF SERVICE DESIGN CONSIDERATIONS

The following section covers other considerations that were considered while implementing the above program.

## 4.1 Security

- **Packet Header validation**: The magic number and version is verified to ensure that the packet is being received from the correct node.
- **Data integrity**: The SHA-1 hash is computed to verify the integrity of data.

## 4.2 Performance

- **Concurrency**: The implementation is highly concurrent without utilizing synchronization, thus ensuring that there are no deadlocks. Also parallel uploads and downloads are supported.
- **Minimal packet overhead**: As the implementation uses UDP, so the packet overhead is low.

## 4.3 Handling crashes

The application is resilient to crash, the timeouts are in place such that it can restart download from another peer.

# 5 INSTALLATION CONSIDERATIONS

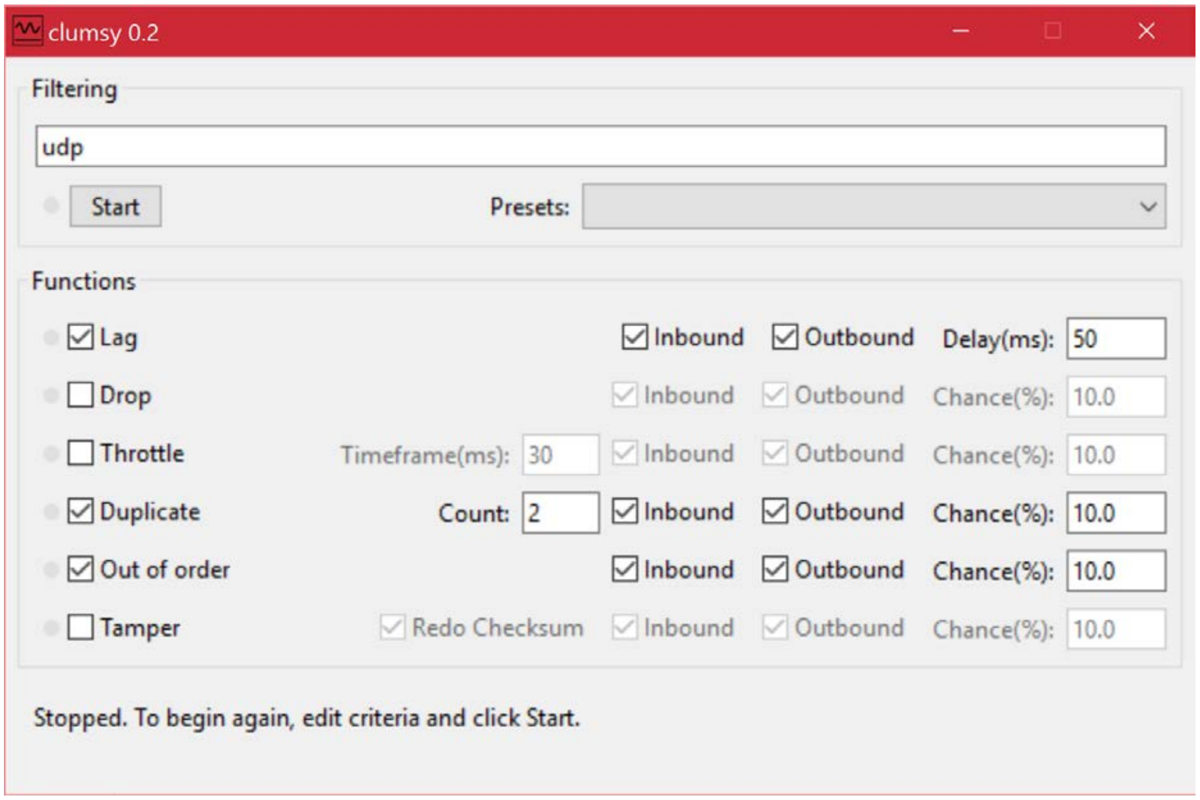
The following steps must be performed:

1. Ensure that java 8 is installed.
2. Ensure that the data is already chunked.
3. Ensure that necessary files Master Data file, has chunks file, node map files are placed in the accessible directory. These are supplied with the zip under the test/files directory.
4. Download and Install clumsy for simulating packet loss, loss etc. Set its option to UDP. https://jagt.github.io/clumsy/
5. Place the jar in the directory that is accessible. The jar is in the test directory of the supplied zip file.
6. Run the command for example java -jar peer.jar -f files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
7. You can open multiple command prompts to run multiple instances of the peer.
8. **All the chunk outputs and problem files are created in the temporary directory of the OS, so files have to be looked up from there**.

# 6 TESTING STRATEGY AND RESULTS

As expected window size varies by congestion, loss and reordering of packets. Clumsy was used to simulate network packet loss, lag, and other scenarios. Various combinations of scenarios were executed and some of them have been mentioned in the next subsection.

The following screenshot shows clumsy in action.



clumsy 0.2

Filtering

udp

Start    Presets:

Functions

| | | | | | |
|---|---|---|---|---|---|
| ✓ Lag | | ✓ Inbound | ✓ Outbound | Delay(ms): | 50 |
| ☐ Drop | | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ☐ Throttle | Timeframe(ms): 30 | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ✓ Duplicate | Count: 2 | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ✓ Out of order | | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ☐ Tamper | ✓ Redo Checksum | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |

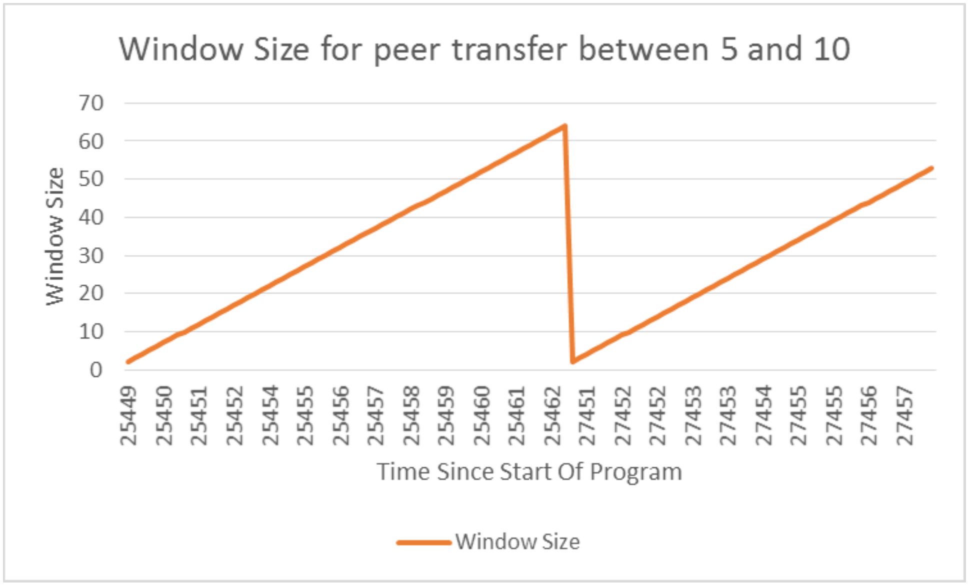
Stopped. To begin again, edit criteria and click Start.

**Figure 2: Clumsy**

## 6.1 Scenarios

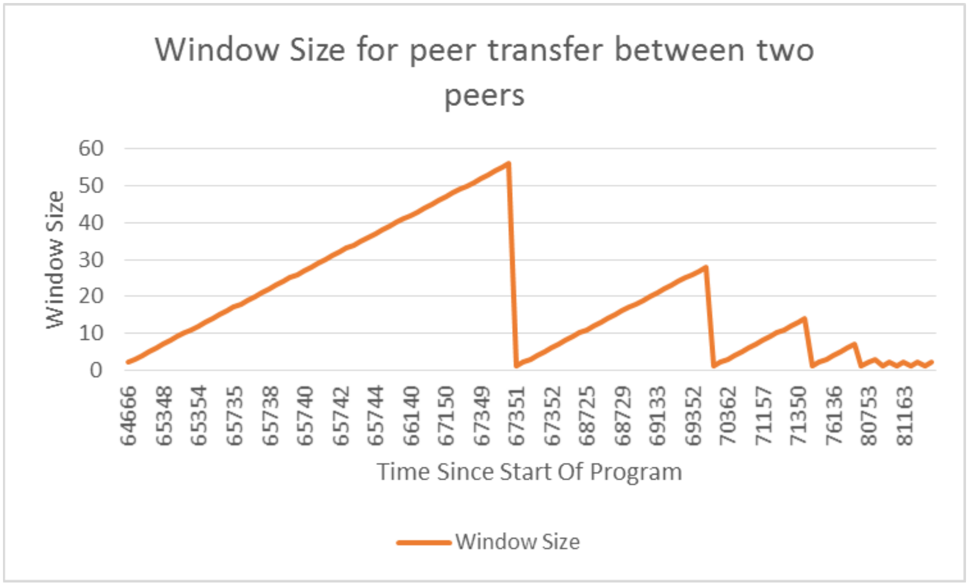
**No packet loss scenario: 2 nodes**

With no loss as expected the window size increases for each chunk till the threshold is reached. Also requests completed at a fast pace.



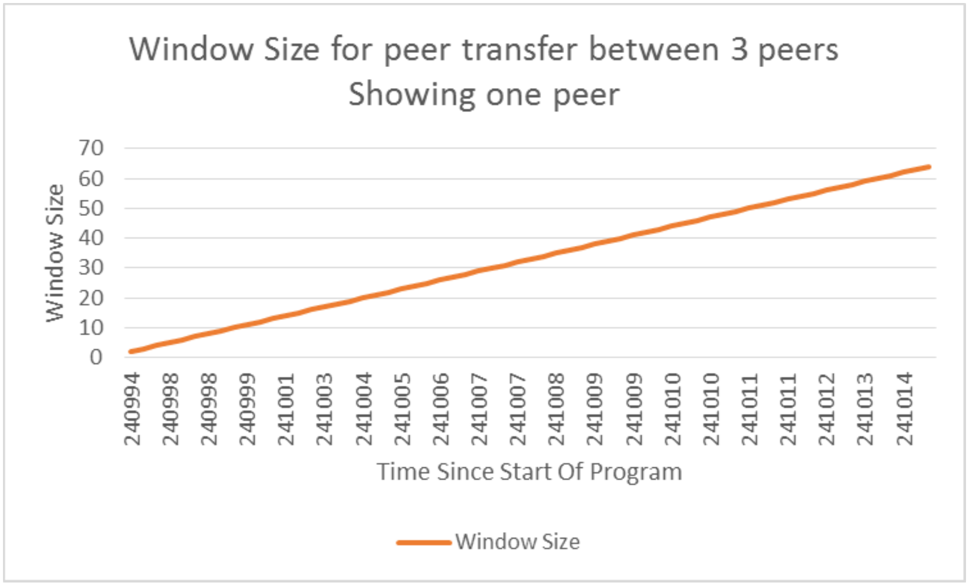
**Packet loss scenario: 2 nodes**

I started clumsy and turned on all the options as shown below. As expected the window size did decrease a lot, signifying packet loss and other issues. In this case the first chunk was transferred before loss came into picture. And as can be seen the performance of the application deteriorated severely by looking at the time taken by the application.



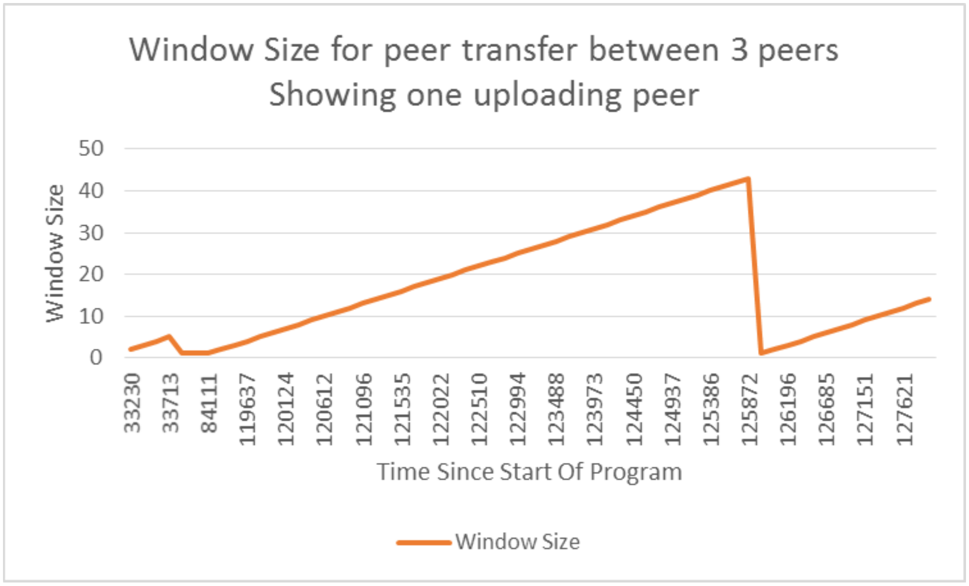
**No packet loss scenario: 2 nodes supporting concurrent download**

As the requests are now distributed the graph shows performance of a single chunk handled by the node. Here two pairs of the nodes are serving each file chunk set, so you get almost twice the performance.

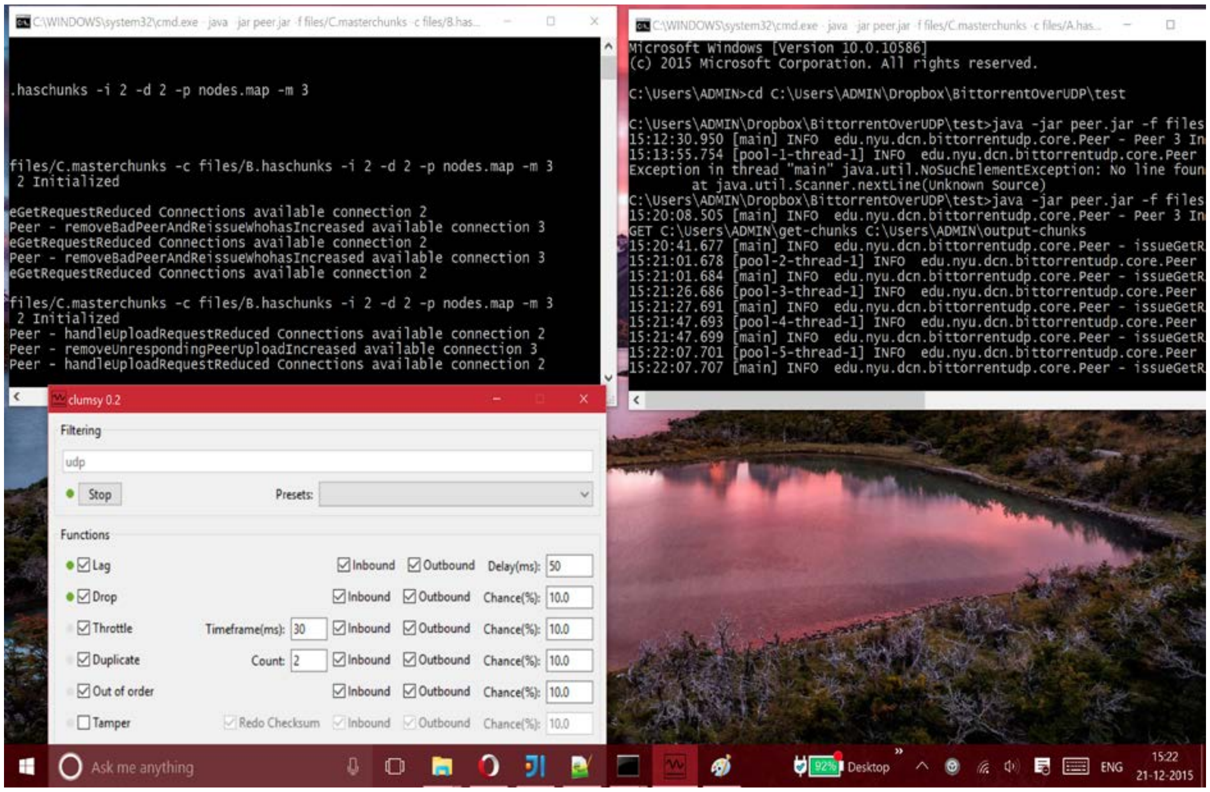


## Packet loss scenario: 2 nodes supporting concurrent download

In this case there was packet loss, and the performance reduced significantly, only when the emulator was turned off did the performance reach satisfactory levels.



## 7 SCREENS


C:\WINDOWS\system32\cmd.exe - java -jar peer.jar -f files/C.masterchunks -c files/B.has...

```
.haschunks -i 2 -d 2 -p nodes.map -m 3



files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
 2 Initialized

eGetRequestReduced Connections available connection 2
Peer - removeBadPeerAndReissueWhohasIncreased available connection 3
eGetRequestReduced Connections available connection 2
Peer - removeBadPeerAndReissueWhohasIncreased available connection 3
eGetRequestReduced Connections available connection 2

files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
 2 Initialized
Peer - handleUploadRequestReduced Connections available connection 2
Peer - removeUnrespondingPeerUploadIncreased available connection 3
Peer - handleUploadRequestReduced Connections available connection 2
```

C:\WINDOWS\system32\cmd.exe - java -jar peer.jar -f files/C.masterchunks -c files/A.has...

```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\ADMIN>cd C:\Users\ADMIN\Dropbox\BittorrentOverUDP\test

C:\Users\ADMIN\Dropbox\BittorrentOverUDP\test>java -jar peer.jar -f files
15:12:30.950 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - Peer 3 In
15:13:55.754 [pool-1-thread-1] INFO  edu.nyu.dcn.bittorrentudp.core.Peer
Exception in thread "main" java.util.NoSuchElementException: No line foun
        at java.util.Scanner.nextLine(Unknown Source)
C:\Users\ADMIN\Dropbox\BittorrentOverUDP\test>java -jar peer.jar -f files
15:20:08.505 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - Peer 3 In
GET C:\Users\ADMIN\get-chunks C:\Users\ADMIN\output-chunks
15:20:41.677 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - issueGetR
15:21:01.678 [pool-2-thread-1] INFO  edu.nyu.dcn.bittorrentudp.core.Peer
15:21:01.684 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - issueGetR
15:21:26.686 [pool-3-thread-1] INFO  edu.nyu.dcn.bittorrentudp.core.Peer
15:21:27.691 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - issueGetR
15:21:47.693 [pool-4-thread-1] INFO  edu.nyu.dcn.bittorrentudp.core.Peer
15:21:47.699 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - issueGetR
15:22:07.701 [pool-5-thread-1] INFO  edu.nyu.dcn.bittorrentudp.core.Peer
15:22:07.707 [main] INFO  edu.nyu.dcn.bittorrentudp.core.Peer - issueGetR
```

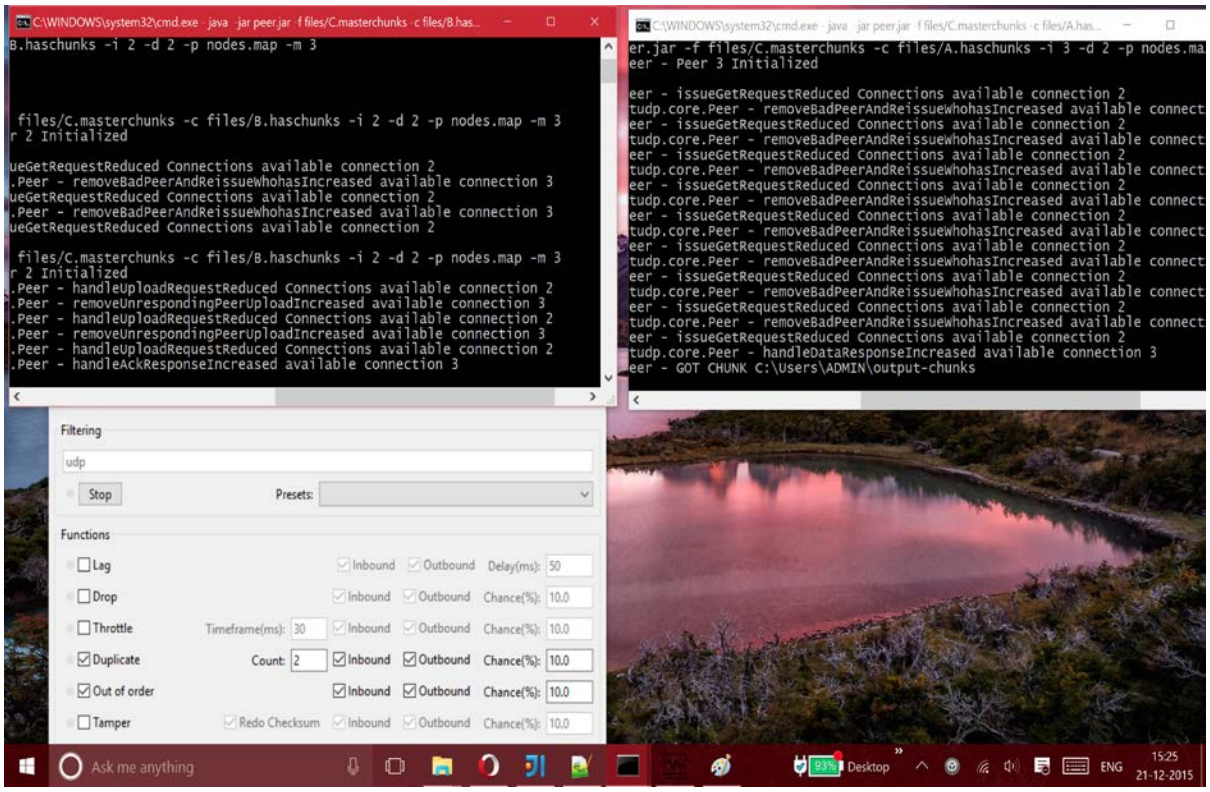
Filtering: udp

Presets:

Functions

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| ● ✓ Lag |  | ✓ Inbound | ✓ Outbound | Delay(ms): | 50 |
| ● ✓ Drop |  | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ✓ Throttle | Timeframe(ms): 30 | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ✓ Duplicate | Count: 2 | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ✓ Out of order |  | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ☐ Tamper | ✓ Redo Checksum | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |

**Figure 3: 2 node scenario with packet loss and clumsy**



```
C:\WINDOWS\system32\cmd.exe - java -jar peer.jar -f files/C.masterchunks -c files/B.has...
B.haschunks -i 2 -d 2 -p nodes.map -m 3



 files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
r 2 Initialized

ueGetRequestReduced Connections available connection 2
.Peer - removeBadPeerAndReissueWhohasIncreased available connection 3
ueGetRequestReduced Connections available connection 2
.Peer - removeBadPeerAndReissueWhohasIncreased available connection 3
ueGetRequestReduced Connections available connection 2

 files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
r 2 Initialized
.Peer - handleUploadRequestReduced Connections available connection 2
.Peer - removeUnrespondingPeerUploadIncreased available connection 3
.Peer - handleUploadRequestReduced Connections available connection 2
.Peer - removeUnrespondingPeerUploadIncreased available connection 3
.Peer - handleUploadRequestReduced Connections available connection 2
.Peer - handleAckResponseIncreased available connection 3
```

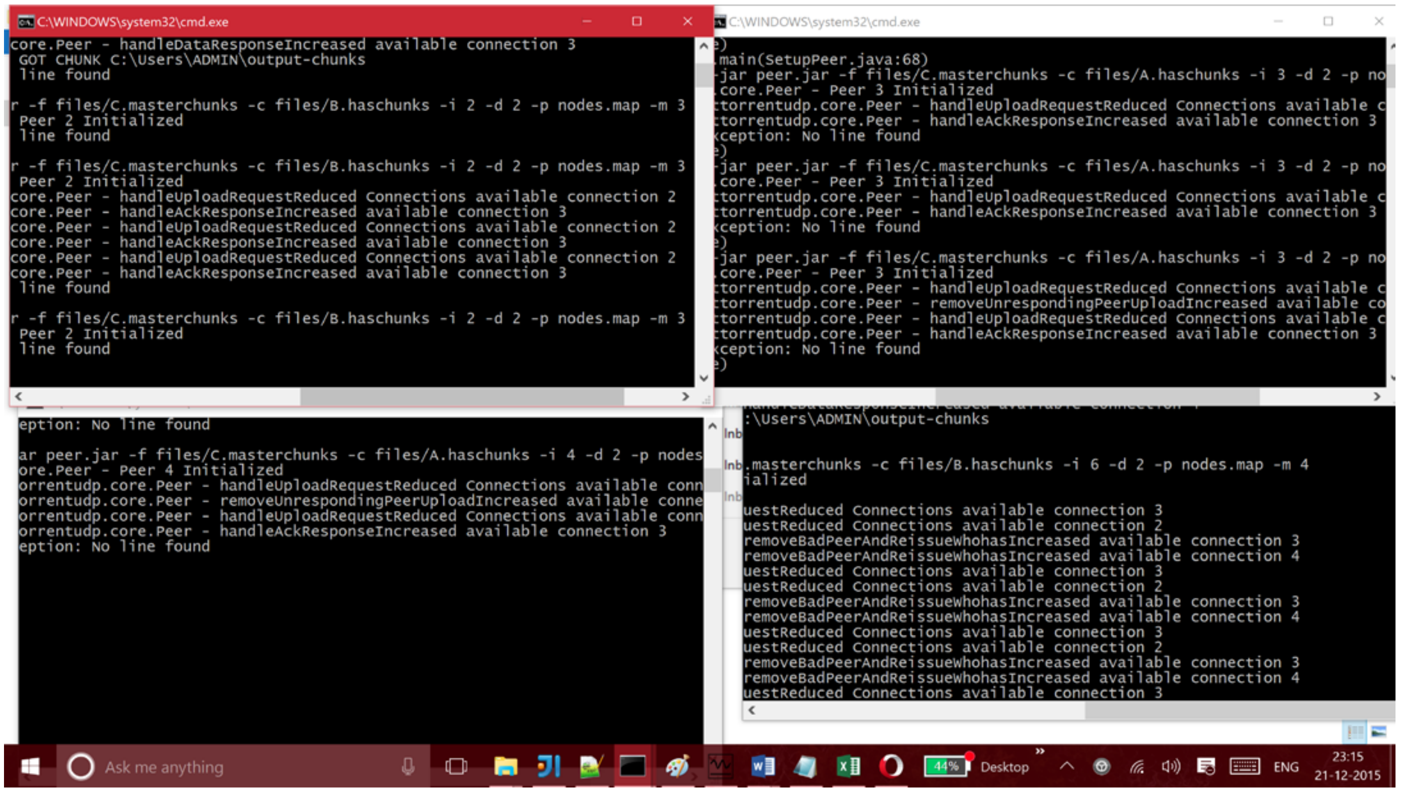
```
C:\WINDOWS\system32\cmd.exe - java -jar peer.jar -f files/C.masterchunks -c files/A.has...
er.jar -f files/C.masterchunks -c files/A.haschunks -i 3 -d 2 -p nodes.ma
eer - Peer 3 Initialized

eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - removeBadPeerAndReissueWhohasIncreased available connect
eer - issueGetRequestReduced Connections available connection 2
tudp.core.Peer - handleDataResponseIncreased available connection 3
eer - GOT CHUNK C:\Users\ADMIN\output-chunks
```

Filtering: udp

| Function | | | | | |
|---|---|---|---|---|---|
| ☐ Lag | | ✓ Inbound | ✓ Outbound | Delay(ms): | 50 |
| ☐ Drop | | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ☐ Throttle | Timeframe(ms): 30 | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |
| ☑ Duplicate | Count: 2 | ☑ Inbound | ☑ Outbound | Chance(%): | 10.0 |
| ☑ Out of order | | ☑ Inbound | ☑ Outbound | Chance(%): | 10.0 |
| ☐ Tamper | ✓ Redo Checksum | ✓ Inbound | ✓ Outbound | Chance(%): | 10.0 |

Figure 4: Clumsy Lag drop and throttle disabled



```
C:\WINDOWS\system32\cmd.exe
core.Peer - handleDataResponseIncreased available connection 3
 GOT CHUNK C:\Users\ADMIN\output-chunks
 line found

r -f files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
 Peer 2 Initialized
 line found

r -f files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
 Peer 2 Initialized
core.Peer - handleUploadRequestReduced Connections available connection 2
core.Peer - handleAckResponseIncreased available connection 3
core.Peer - handleUploadRequestReduced Connections available connection 2
core.Peer - handleAckResponseIncreased available connection 3
core.Peer - handleUploadRequestReduced Connections available connection 2
core.Peer - handleAckResponseIncreased available connection 3
 line found

r -f files/C.masterchunks -c files/B.haschunks -i 2 -d 2 -p nodes.map -m 3
 Peer 2 Initialized
 line found
```

```
C:\WINDOWS\system32\cmd.exe
e)
.main(SetupPeer.java:68)
-jar peer.jar -f files/C.masterchunks -c files/A.haschunks -i 3 -d 2 -p no
.core.Peer - Peer 3 Initialized
ttorrentudp.core.Peer - handleUploadRequestReduced Connections available c
ttorrentudp.core.Peer - handleAckResponseIncreased available connection 3
xception: No line found
e)
-jar peer.jar -f files/C.masterchunks -c files/A.haschunks -i 3 -d 2 -p no
.core.Peer - Peer 3 Initialized
ttorrentudp.core.Peer - handleUploadRequestReduced Connections available c
ttorrentudp.core.Peer - handleAckResponseIncreased available connection 3
xception: No line found
e)
-jar peer.jar -f files/C.masterchunks -c files/A.haschunks -i 3 -d 2 -p no
.core.Peer - Peer 3 Initialized
ttorrentudp.core.Peer - handleUploadRequestReduced Connections available c
ttorrentudp.core.Peer - removeUnrespondingPeerUploadIncreased available co
ttorrentudp.core.Peer - handleUploadRequestReduced Connections available c
ttorrentudp.core.Peer - handleAckResponseIncreased available connection 3
xception: No line found
e)
```

```
eption: No line found

ar peer.jar -f files/C.masterchunks -c files/A.haschunks -i 4 -d 2 -p nodes
ore.Peer - Peer 4 Initialized
orrentudp.core.Peer - handleUploadRequestReduced Connections available conn
orrentudp.core.Peer - removeUnrespondingPeerUploadIncreased available conne
orrentudp.core.Peer - handleUploadRequestReduced Connections available conn
orrentudp.core.Peer - handleAckResponseIncreased available connection 3
eption: No line found
```

```
:\Users\ADMIN\output-chunks


.masterchunks -c files/B.haschunks -i 6 -d 2 -p nodes.map -m 4
ialized

uestReduced Connections available connection 3
uestReduced Connections available connection 2
removeBadPeerAndReissueWhohasIncreased available connection 3
removeBadPeerAndReissueWhohasIncreased available connection 4
uestReduced Connections available connection 3
uestReduced Connections available connection 2
removeBadPeerAndReissueWhohasIncreased available connection 3
removeBadPeerAndReissueWhohasIncreased available connection 4
uestReduced Connections available connection 3
uestReduced Connections available connection 2
removeBadPeerAndReissueWhohasIncreased available connection 3
removeBadPeerAndReissueWhohasIncreased available connection 4
uestReduced Connections available connection 3
```

Figure 5: Multiple downloads with loss

# 8 CONCLUSIONS

## What went well

- Implementation is highly concurrent and fault tolerant.
- Implementation does not use synchronization constructs, yet is highly concurrent. Only resource to whom access is synchronized is the socket object.

## What went not so well

- Code could be optimized further, since the time to implement was limited so not all optimizations could be made.