Executive Summary

The development of BitTorrent clients requires constant testing with a swarm of other peers. This is both a tedious and dangerous process if connecting to real peers and trackers over the Internet as BitTorrent is designed to do. For example, if the client in development has software bugs, they could potentially damage members of the swarm, each of which is running on dedicated hardware.

The solution is to emulate a swarm so that the client being developed never connects to external hardware. This will also allow for rapid development as the swarm can be modified to behave in ways that highlight particular features being tested. For instance, the behaviour of peers can be modified so that they respond slowly or return bad responses to requests. In this case the client may want to prioritise other peers or drop the connection to bad peers.

By virtualising the swarm, the system is also able to take liberties in the internal communications to simply and optimise communication. In reality, each peer would be communicating with each other, but because the client is the only thing being tested, this communication is unnecessary. Instead simply increasing the amount of pieces that a virtual peer can connect to during runtime can effectively emulate peer communication with much less overhead.

Future extensibility has also been designed into the proposed solution. The Network Interface allows for multiple clients to be connected at once, peers can be updated and added dynamically during runtime and peer behaviour can be modified so that the swarm's dynamic changes. All these changes can me made quickly and in extreme amounts, allowing the client to be tested in all situations.

This document covers the object design for the problem analysis, class selection and responsibilities, bootstrap process and a basic verification. It does not attempt to show implementations of the candidate components. It also does not attempt to implement many features found in specifications of the BitTorrent protocol newer that 1.0.

Table of Contents

1 Problem Analysis

1.1 Goals

The creation of an emulated swarm that an external client can connect to. This external client must be able to communicate with the emulated swarm as if it were real.

The emulated swarm must be able to emulate different behaviours common to real swarms so that the client can be tested under varying circumstances.

1.2 Assumptions

The following assumptions have been made in order to create a complete object design:

A1. A torrent file exists with all relevant information pertaining to the 'swarm'. It will contain an IP address and port corresponding the virtual tracker.

A2. A directory exists that contains all the files needed for the transfer in locations as specified by the torrent file.

A3. A configuration file exists the containing at least
   a. The number of peers to initialise
   b. The unique ID, IP address, port and the pieces for each peer

A4. *NetworkInterface* is a class that has been implemented to act as a virtual router. This class will accept incoming traffic and convert IP address and port into an appropriate format. The incoming message will be encapsulated and forwarded to the correct destination within the emulated swarm. *NetworkInterface* will also conform to the responsibilities listed in its CRC Card

A5. *NetworkInterface* will do the reverse for messages from the peers and tracker to the client.

A6. The client is a separate system and outside the scope of emulation. It will attempt to behave as if it were connected via a router to an external network of peers and a tracker. Thus, any faulty messages received can be assumed to be errors in the client.

A7. The emulated swarm will not need to conform to later versions of the Bittorrent specification after 1.0

A8. The system used will be able to handle the amount of emulated peers required.

A9. Only one tracker will be used.

A10. Peers cannot have their pieces removed once added.

1.3 Simplifications

The emulated swarm will only appear to act as a Bittorrent swarm to external clients, internally it is simplified in the following ways:

- Peers contain an integer index of each piece they contain. To retrieve the data, the file object must be interrogated. Peers cannot communicate with pieces in any way.
- Peers can have behaviours that allow them to act differently to each other. These behaviours only affect *Messages* that have already been generated, ready for return.
- Neither Peers or the Tracker initiate messages to the client. They also do not communicate with each other in any way. If a peer is to be updated with new 'pieces' then its piece index collection is simply appended with the new information.
- As there is no concept of peers actively communicating with each other, then there is also no concept of seeds and leechers within the emulated swarm. This means that the emulated swarm does not contain logic to actively change the overall speed of the swarm as would happen depending on the natural seed/leecher ratio in reality.
- Although peer and tracker share some common information and methods, inheritance has not been used to generalise their behaviour as there is a limit of one tracker in the system and as such, the complexities introduced would outweigh any benefits.

2 Candidate Classes

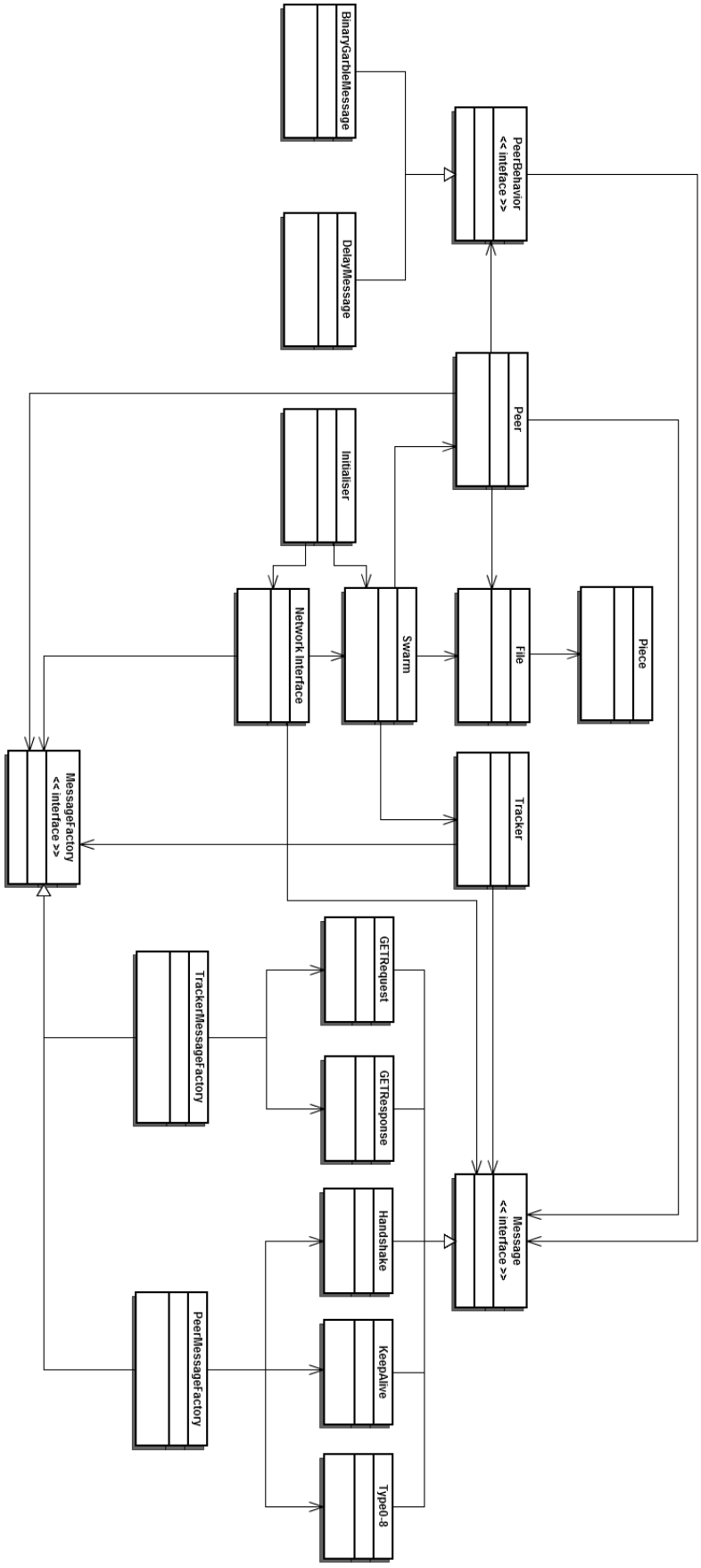
2.1 Overview

The classes that contribute to the proposed solution are described below. Note that all UML diagrams follow the conventions defined in the document references in Section 6.

2.1.1 Candidate Class List

- Initialiser
- NetworkInterface
- Swarm
- MessageFactory
    - TrackerMessageFactory
    - PeerMessageFactory
- File
- Piece
- Peer
- PeerBehaviour
    - MessageDelay
    - MessageBinaryGarble
- Tracker
- Message
    - GETRequest
    - GETResponse
    - Handshake
    - KeepAlive
    - Type0-8

2.1.2 Class Diagram



2.1.3 Justification

The system is designed to emulate a BitTorrent swarm only in the sense of input and output from the swarm.

Pieces are only directly interacted with via *File*. This allows the design to avoid circular dependencies. This also means that *Peers* simply have an index that is passed to *File*, allowing for easy updating of the virtual pieces that *Peer* can appear to contain.

By externalising the behaviours of the *Peers*, the system allows for each peer having a different set of behaviours. This is not needed for *Tracker,* as there is only one tracker. It could be implemented to have a series of internal behaviours but these have not been separated because it would add unneeded complexity.

Message Factories have been included so that the objects creating the messages do not need to be able to access data that is required to create a message. It also separates the creation of a message from its usage to reduce the responsibility of components. The factory will take care of choosing the type of *Message* to be created.

Tracker is kept completely separate from *Peers* because it only needs to be able to return a list of IP addresses and Ports. The *Peers* do not need a concept of their IP address or port.

2.2 CRC Cards

2.2.1 NetworkInterface

NetworkInterface handles buffering of incoming/outgoing network traffic, it encapsulates and de-encapsulates data to/from *Message* instances. It will also support the connection of multiple external clients by forwarding any incoming network traffic that is not destined to an internal peer or tracker back to the external network.

Component: NetworkInterface	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Encapsulate Incoming Network Traffic to *Messages*	TrackerMessageFactory PeerMessageFactory
Forward *Messages* to the *Swarm*	Swarm
Send data from returned *Messages* out onto the Network	n/a
Forward incoming network traffic to external source if not targeting internal 'IP address and port'	n/a

2.2.2 Peer

Peer is the virtualisation of a peer in the swarm. It is able to accept and respond to incoming *Messages* from/to the *Swarm* component. Peers will be held in some form of container within Swarm, so that they can be added and removed dynamically in later versions.

Peer will respond to handshake messages but will not initiate them.

When a request for a block is received, an instance of *Peer* will ask *File* to locate the appropriate binary data by passing it a piece and block index. This data is then passed back to *Swarm* as an encapsulated *Message*, built by Peer*MessageFactory*.

A *Peer* instance can be affected by chosen *PeerBehaviours* that implement a Strategy Pattern. In this way, *Peer* can return data that has been changed in some way, perhaps delayed or garbled, or simply not reply. Before passing a *Message* to *Swarm*, *Peer* will send it to get changed or delayed (for example) as a *Peerbehaviour.*

Component: Peer	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Respond to Handshake Message	*MessageFactory* *PeerBehaviour* *Message*
Respond to KeepAlive Message	*MessageFactory* *PeerBehaviour* *Message*
Respond to Block (Type0-8) Request Message	*MessageFactory* *PeerBehaviour* *File* *Message*
Create chosen *PeerBehaviour*	*PeerBehaviour*

2.2.3 PeerBehaviour

PeerBehaviour is an interface to behaviours that can be adopted by an instance of *Peer*, based on the Strategy Pattern.

Note: MessageBinaryGarble and MessageDelay are just two examples of possible behaviours. They are not given their own CRC cards, as their design is not in the scope of this project.

Component: PeerBehaviour	
SubClasses: MessageBinaryGarble MessageDelay	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Modify *Messages* to allow *Peers* to behave more realistically.	*Message*

2.2.4 Swarm

Swarm is responsible for creating and initialising the *Peer, Tracker and File* instances. *Messages* to and from *NetworkInterface* are passed through swarm and it is ultimately *Swarm,* that forwards the message to the appropriate *Peer* or *Tracker.*

Swarm has the ability to dynamically add and remove *Peers* during runtime. When a *Peer* is created or removed, the appropriate IP address and port information must also be added or removed from *Tracker*.

Swarm can also update the *Peers* or *Tracker* at any time. This could be to change the piece/block indexes stored in a *Piece* or to add or remove IP addresses and port information from *Tracker.* (Possibly leading to a situation where the client is returned an IP address to a virtual peer that does exist).

NetworkInterface calls the functions of *Swarm* directly.

Component: Swarm	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Forward GETRequest Message to Tracker	*Tracker* *Message*
Forward Handshake Message to Appropriate Peer	*Peer* *Message*
Forward KeepAlive Message to Appropriate Peer	*Peer* *Message*
Forward BlockRequest(Type0-8) Message to Appropriate Peer	*Peer* *Message*
Create and Initialise Peer Objects	*Peer* *File* *Tracker*
Update Peer Object	*Peer*
Create and Initialise Tracker Object	*Tracker*
Update Tracker Object	*Tracker*
Create and Initialise File Object	*File*

2.2.5 File

File is a smart caching object. *File* has a collection of *Pieces* that it creates during the bootstrap process. *File* can accept requests for binary data of a particular block and piece index and will request this data from the appropriate *Piece.* If the *Piece* does not have any data, *File* will allocate binary data to it.

In this way, *File* can keep track of how much binary data is loaded into memory at any given time and ask *Piece* to delete the data it stores if it is no longer needed.

Component: File	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Create collection of *Pieces*	Piece
Read file on disk and allocate binary data to *Pieces* as needed.	Piece
Memory management of loaded binary data	Piece

2.2.6 Piece

Piece exists only as a way to avoid loading an entire file into memory at once. It is given data to encapsulate by *File* and is able to delete it or return a chunk (block) of data if asked by *File*.

Component: Piece	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Encapsulate data	*n/a*
Delete data	*n/a*
Retrieve a block from a piece of data	*n/a*

2.2.7 Tracker

Tracker contains a list of the IP address and ports of all the peers in the emulated swarm. *Tracker* has no concept or connection to the *Peer* component.

In future versions, the response could be delayed or garbled in a similar way to the *Peer* messages. These would not need to be implemented as behaviours because there is only one *Tracker*.

Component: Tracker	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Respond to GETRequest	*MessageFactory* *Message*

2.2.8 Message

Message is the interface to the specific types of messages. Each will encapsulate data in a different way.

Message exists so that messages can be passed through the system by components with no knowledge of the subclasses.

Ultimately, *Messages* will be passed from *NetworkInterface* to either *Peer* or *Tracker*. They will be acted upon and new *Messages* generated to be eventually passed back to *Network*k*Interface*.

Note: the subclasses have not been given individual CRC cards. Each will have the same responsibility as *Message* for its own format of data.

Component: Message	
SubClasses: GETRequest GETResonse KeepAlive Handshake Type0-8	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Encapsulate Data	n/a

2.2.9 MessageFactory

MessageFactory is the interface to *TrackerMessageFactory* and *PeerMessageFactory*. They have the responsibility of creating *Message* objects to encapsulate data.

Component: MessageFactory	
SubClasses: TrackerMessageFactory PeerMessageFactory	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Generate Message Objects	*Message* *Peer/Tracker (depending on implementation)* *Swarm*

2.2.10 Initialiser

Initialiser has the task of reading the configuration file then creating and initialising the above components where needed. See Section 4 for the bootstrap process that includes the initialisation process.

Component: Initialiser	
SubClasses: n/a	**SuperClasses:** n/a
Responsibilities	**Collaborators**
Read and parse configuration File	*n/a*
Create and Initialise Swarm Object	*Swarm*
Create and initialise *TrackerMessageFactory*	*TrackerMessageFactory*
Create and initialise *PeerMessageFactory*	*PeerMessageFactory*
Create and initialise *NetworkInterface*	*NetworkInterface*

3 Quality

3.1 Design Patterns

The following design patterns have been implemented:

- Strategy: *PeerBehaviour* is a interface to specific behaviours that can be given to a peer. This allows each peer to be given different ways to respond to *Messages*.
- Factory Method: *MessageFactory* separates the job of creating *Message* objects from using them. This means that

3.2 Extensions

The system is designed so that it can be easily extended in future iterations in the following ways:

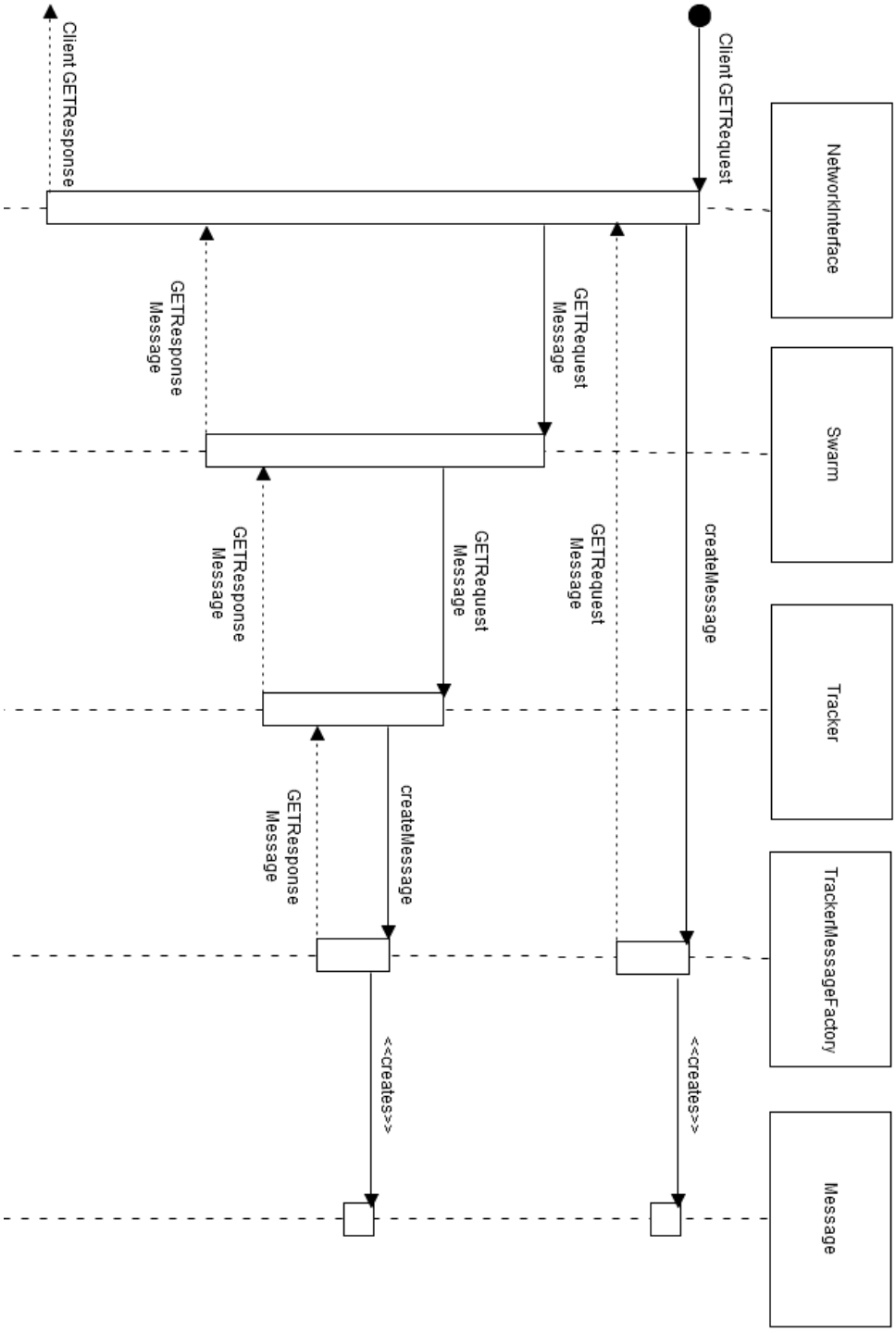
- The strategy pattern utilised in *PeerBehaviour* means that peers can be assigned different ways to alter the return of a message. These alterations could include delaying the return (something likely wanted in most cases to emulate internet latency), binary data mangling, dropping the message entirely and many others.
- *Peers* are created in a container so that they can be dynamically created and removed during runtime.
- *Swarm* has access to *Peer* and *Tracker* components. It can therefore alter their properties during runtime. This could include updating *Tracker* to include the new IP addresses of added *Peers* or causing it to return IP addresses of non-existent virtual peers.
- *File* has the ability to ask *Piece* to delete its stored binary data. This means that only parts of the file need to be loaded into memory at any given time. This allows for caching algorithms such as putting the rare pieces into memory as clients will tend to ask for these first. Also, if more than one client is connected to the virtual swarm, then this caching should speed up transfers.
- It is likely that during testing, multiple clients will be connected to the emulated swarm. *NetworkInterface* is designed to forward to appropriate locations so that more than one client is supported. This could be useful if the client's developers need to know how the client will react to a specific competitor's client, which cannot be virtualised accurately.

4 Bootstrap Process

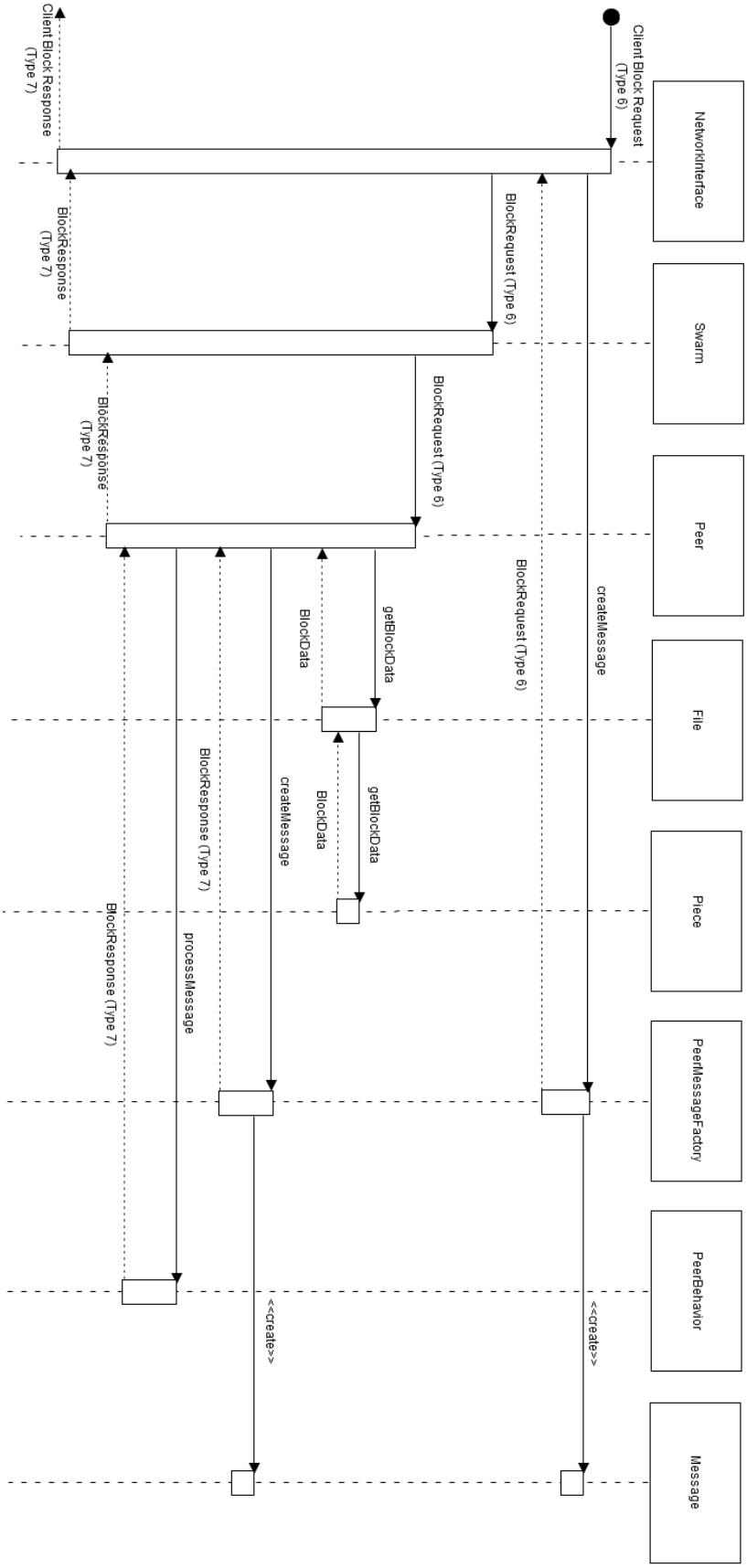
1. Mainline creates *Initialiser* instance.
2. *Initialiser* reads configuration file from disk.
3. *Initialiser* parses configuration data into local variables.
4. *Initialiser* creates *TrackerMessageFactory* instance.
5. *Initialiser* creates *PeerMessageFactory* instance.
6. *Initialiser* creates *Swarm* instance.
7. *Swarm* creates *File* instance.
8. *File* creates collection of *Piece* instances.
9. *Swarm* creates *Tracker* instance.
10. *Swarm* creates collection of *Peer* instances and updates tracker with new peers information.
11. Peer creates appropriate *PeerBehaviour.*
12. Initialiser creates *NetworkInterface*.
13. Initialiser starts *NetworkInterface* execution.

5 Verification

5.1 Request To Tracker For Peer List



5.2 Request For Block



6 References

UML Superstructure Specification Version 2.4
http://www.omg.org/spec/UML/2.4/

A Technical Description of the BitTorrent Protocol
http://www.cse.chalmers.se/~tsigas/Courses/DCDSeminar/Files/BitTorrent.pdf