CS3017 Assessment: A Power-Law Peer-to-Peer System

Philip Hale

November 27, 2013

1 Peer Behaviour

The system consists of three types of agent: Super Peers, Ordinary Peers and a single Host Cache. The Host Cache is implemented as an agent in order to unify the connection behaviour between elements of the system. In other words, peers can connect to the Host Cache using the same mechanisms that are required to connect to each other.

It can be helpful to think of the identity of an agent as defined by the kinds of behaviours it can support. What we mean here by a behaviour is the ability to respond in some way to a particular situation or event. Some behaviours continually monitor the state of the agent in the network, whereas others are triggered by incoming messages from other agents.

The process of specifying and identifying different messages is explained in another section.

Super Peers and Ordinary Peers implement a common set of behaviours. For the sake of simplicity, the designation of peers into Super and Ordinary sets will occur during initialization of the system.

1.1 Host Cache

The Host Cache is the entry-point for the network with two responsibilities. Firstly, to maintain a list of connected peers. Secondly, to supply peers with a list of Super Peers which they can attempt to connect to

Theses behaviours can be summarised by the following piece of pseudocode. It will run when the Host Cache receives a message identified as 'Neighbours Request'.

1.1.1 Receive 'Neighbours Request'

Algorithm 1: Receive 'Neighbours Request'

input: Message Name: 'Neighbours Request'
Message Sender: *peer*
Data: *peerList*: A hash of peers with value true if peer is super

```
1 if peer list doesn't contain peer then
2     add peer to peerList;
3 for peer ∈ peerList do
4     if peerList[peer] = true then
5         neighbours ← peer;
6 reply('Neighbours Response', neighbours)
```

1.2 Peers

Here we will explain the different types of behaviours implemented by both Super and Ordinary Peers. Some of these behaviours are shared between all peers, but their effects will differ depending on whether the peer is Super or Ordinary. This is a design decision which compromises simplicity of the described behaviours with simplicity of the system as a whole.[1]

[1]The idea is analogous to the contradiction between two rules of software engineering: The Law of Demeter which aims to minimise method chaining, and the idea of class cohesion which

This behaviour is responsible for maintaining a list of Super Peers that the peer can connect to. One consequence of this design is that the peers will always have a list of potential peers to connect to, even when they already have their maximum number of connected peers. Additionally, $knownPeers \cap connectedPeers \neq \emptyset$.

Algorithm 2: Send 'Neighbours Request'

Data: *knownPeers*: A list of neighbours known but not connected to the peer
Data: *hostCache*: Address of the Host Cache
1 **if** $knownPeers = \emptyset$ **then**
2 `send(`*hostCache, 'Neighbours Request'*`)`

Algorithm 3: Receive 'Neighbours Response'

input: Message Name: 'Neighbours Response'
Neighbours: *peers*
Message Sender: *sender*
Data: *knownPeers*: A list of neighbours known but not connected to the peer
Data: *hostCache*: Address of the Host Cache
1 **if** $sender = hostCache$ **then**
2 $knownPeers \cap peers$;

Algorithm 4: Send 'Connect Request'

Data: *knownPeers*: A list of neighbours known but not connected to the peer
Data: *connectPendingPeers*: Peers that have been sent a connection request
Data: *minPeers*: Required number of connected peers.
1 **if** $knownPeers \neq \emptyset$ *and*
`length(`$connectPendingPeers \cap connectedPeers$`)`$< minPeers$ **then**
2 `send(`$\exists p \in knownPeers$, *'Connect Request'*`)`

1.3 Message Definitions

- Neighbours Request
- Neighbours Response
- Search Request
- Search Response
- File Request
- File Response
- File List
- Connect Request

gives an object well-bounded behaviour.

- Connect Response