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.

In these examples, some simplifications have been made. For example, we ignore the difference between a file and hash of the file which identifies it. When an algorithm states that it sends a file, it can be assumed that it is sending a hash of the file. In addition, this masks the fact that a peer will have a hash table of some sort in order to identify the file from the file name.

Assumptions have been made that connections will not be lost, and that messages will only be sent by peers that should send them.

Contents

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'

```
input: Sender: peer
       Message Name: 'Neighbours Request'
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 $peer \in peerList$ do
4     if $peerList[peer] = true$ then
5         $neighbours \leftarrow neighbours \cup peer$;
6 send(peer, 'Neighbours Response', neighbours)
```

1.2 All 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]

1.2.1 Send 'Neighbours Request'

The first message a peer will send: asks the Host Cache for a list of peers which it can connect to in order to join the network. In addition, this message is sent whenever the knownPeers list is empty. This ensures that the peer always has some Super Peer address to connect to, in the event that they need more connections.

```
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')
```

1.2.2 Receive 'Neighbours Response'

A reply to the 'Neighbours Request' message. The message payload contains a list of Super Peers which the peer can try and connect to.

[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 gives an object well-bounded behaviour.

```
  input: Message Name: 'Neighbours Response'
         Potential Neighbours: $peers$
  Data: $knownPeers$: A list of Super Peers known but not connected to
        the peer
1 $knownPeers \leftarrow knownPeers \cup peers$;
```

1.2.3 Send 'Connect Request'

Asks a Super Peer to be a connection. In order to prevent blocking, the contacted Super Peer is placed in a list while we wait for a response.

The peer will keep trying to connect to more peers until the number of connected peers is greater than a given minimum. In order to limit on how many connection attempts are made, the count includes peers which we are waiting on for a reply.

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

1.2.4 Receive 'Connect Response'

A connection response is either successful or unsuccessful. An alternative approach would only send a response to the request if it is successful, but this increases the complexity of the behaviour and so has not been adopted.

A connection response from a peer removes that peer from the known peers list. If the connection is successful, the peer is also added to the list of connected peers.

```
  input: Sender: $sender$
         Message Name: 'Connect Response'
         Connection Successful?: $connectSucceed$
  Data: $knownPeers$: A list of Super Peers known but not connected to
        the peer
  Data: $connectPendingPeers$: Super Peers that have been sent a
        connection request
  Data: $connectedPeers$: Super Peers that this peer is connected to
1 $knownPeers \leftarrow knownPeers \setminus \{sender\}$;
2 $connectPendingPeers \leftarrow connectPendingPeers \setminus \{sender\}$;
3 if $connectSucceed = true$ then
4     $connectedPeers \leftarrow connectedPeers \cup \{sender\}$;
```

For this demonstration, a peer will always and continually search for files they want.

1.2.5 Send 'Search Request'

```
   Data: $connectedPeers$: Super Peers that this peer is connected to
   Data: $wantedFiles$: List of files the peer wants but does not have
1  if $wantedFiles \neq \emptyset$ and $connectedPeers \neq \emptyset$ then
2      send($\exists p \in connectedPeers$, 'Search Request', $\exists f \in wantedFiles$)
```

1.2.6 Receive 'File Request'

Since our simulation does not send actual files, at this point we can simply return a token response to indicate that we are transferring the file. In practice, we would have to identify the file from the given identifier, and check it exists before returning etc.

```
   input: Sender: $peer$
          Message Name: 'File Request'
          File: $file$
   Data: $sharedFiles$: List of files the peer wants but does not have
1  send($peer$, 'File Response', 'Here is your file')
```

1.2.7 Receive 'File Response'

This completes the procedure for acquiring files in the network.

```
   input: Message Name: 'File Response'
          File: $file$
   Data: $sharedFiles$: List of files the peer wants but does not have
   Data: $transferringFiles$: List of files that are waiting to be sent to this
          peer.
   Data: $wantedFiles$: List of files the peer wants but doesn't have
1  if $file \in transferringFiles$ then
2      $transferringFiles \leftarrow transferringFiles \setminus \{file\}$;
3      $wantedFiles \leftarrow wantedFiles \setminus \{file\}$;
4      $sharedFiles \leftarrow sharedFiles \cup \{file\}$;
```

1.3 Ordinary Peers

These behaviours are exclusively implemented by Ordinary Peers.

1.3.1 Send 'File List'

An Ordinary Peer is required to submit its list of shared files to its connected Super Peer. This is so that other peers can search for files.

Since this behaviour only applies to Ordinary Peers, we can make the simplifying assumption that the list of connected peers only contains a single peer.

We also ignore the possibility that a SuperPeer can become disconnected, which is not a requirement at this stage.

One further assumption is made. We presume that the list of file hashes that are sent to the Super Peer is small enough to fit in a single message. In practice, this is not likely to be the case.

```
  Data: connectedPeer: Super Peer that this peer is connected to
  Data: sharedFiles: List of files the peer wants but does not have
1 send(connectedPeer, 'File List', sharedFiled)
```

1.3.2 Receive 'Search Response'

When a peer receives a search result, the nature of the protocol means that this must be a successful result. If there is no results found, then there is simply no message returned. The payload of the message includes the particular file that was found and the ID of the peer who owns it. Since in our simulation we do not need to present results in any way, once the result is found we can contact the peer directly to request they transfer the file.

```
  input: Message Name: 'Search Response'
         Result: $file$
         Peer With File: peer
  Data: transferringFiles: List of files that have been requested from
        another peer, and are waiting to be sent to this peer.
  Data: wantedFiles: List of files the peer wants but doesn't have
1 $wantedFiles \leftarrow wantedFiles \setminus \{file\}$;
2 send(peer, 'File Request', file)
  $transferringFiles \leftarrow transferringFiles \cup \{file\}$;
```

1.4 Super Peers

These behaviours are exclusively implemented by Super Peers.

1.4.1 Receive 'Connection Request'

input: a request from a peer to join; output: connection response (y/n)

Super Peers maintain a lit of peers which are connect to them, and the files they are sharing. The first step in this interaction is for the Peer to associate itself with the Super Peer.

```
  input: Sender: peer
         Message Name: 'Connection Request'
  Data: connectedOrdinaryPeers: Ordinary Peers that this peer is
        connected to
  Data: ordinaryPeerLimit: Maximum number of Ordinary Peers this
        Super Peer can connect to
1 if $\|connectedOrdinaryPeers\| < ordinaryPeerLimit$ then
2     $connectedOrdinaryPeers \leftarrow connectedOrdinaryPeers \cup \{peer\}$;
3     send(peer, 'Connect Response', true)
4 else
5     send(peer, 'Connect Response', false)
```

1.4.2 Receive 'File list'

The Super Peer stores the list of files for each Ordinary Peer it is connected to. This is so that it is able to handle search requests.

The files are stored as a hash that maps $fileName \rightarrow peerList$.

```
input: Sender: $peer$
       Message Name: ‘File List’
       Shared Files: $fileList$
Data: $connectedOrdinaryPeers$: Ordinary Peers this peer is connected
      to
Data: $ordinaryPeersFiles$: All shared files of connected Ordinary Peers
for $file \in fileList$ do
    $ordinaryPeersFiles[file]$ $peer$;
```

1.4.3 Receive ‘Search Response’

Super Peers must behave slightly differently when receiving a Search Response message. Unlike for Ordinary Peers, sometimes this response may not be for them. This is because search results do not get delivered directly: they are passed back through intermediary peers using the Document Routing system.

It is worth noting that in our *send*() function we pass as many arguments as we have attributes. In practice, the attributes would be wrapped into a single argument.

```
input: Message Name: ‘Search Response’
       Result: $file$
       Peer With File: $peer$
       Previous Senders of the Search Response: $senderStack$
if $senderStack = \emptyset$ then
    inherit the behaviour of Ordinary Peer;
$superPeer \leftarrow senderStack.pop()$;
send($superPeer$, ’Search Response’, $file$, $peer$, $senderStack$)
```

1.4.4 Receive ‘Search Request’

When a Super Peer receives a Search Request for a file, it first checks its list of files. If the file is found, it replies with a Search Response.

If it is not found, it must pass on the message. If the *senderStack* is empty then the sender must be an ordinary peer, so the sender is added to it. Next, it adds itself to the *senderStack* and passes on the message to $\exists peer \in connectedOrdinaryPeers$.

The peer chosen to receive this message is picked based on the proximity of their ID when compared to the file ID. The details of this operation are not explained in this section, but it introduces a requirement of the implemented system for a consistent hashing algorithm.

```
input: Sender: $peer$
       Message Name: ‘Search Request’
       Requested File: $file$
       Previous Senders of the Search Response: $senderStack$
Data: $connectedOrdinaryPeers$: Ordinary Peers connected to this
      Super Peer
Data: $ordinaryPeersFiles$: All shared files of connected Ordinary Peers
Data: $self$: This peer
if $ordinaryPeersFiles$ contains $file$ then
    $peerWithFile \leftarrow \exists p \in ordinaryPeersFiles[file]$;
    if $senderStack = \emptyset$ then
        send($peer$, ‘Search Response’, $file$, $peerWithFile$)
    else
        send($senderStack.pop()$, ‘Search Response’, $file$, $peerWithFile$,
        $senderStack$)
else
    /* We don’t have the file; forward request to neighbour */
    $nearestNeighbour \leftarrow$ nearest($senderStack$, $file$);
    $senderStack \leftarrow senderStack \cup \{self\}$;
    send($nearestNeighbour$, ‘Search Request’, $file$, $senderStack$)
```

2 Message Definitions

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