2016/17

Tracker BitTorrent Project – Deliverable 3

Aitor De Blas Granja / Kevin Cifuentes Salas

Advanced Distributed Systems / Group 004

13-12-2016

Index

Interval selection mechanism

To be honest, we firstly think and implement a hardcoded value for the interval attribute at AnnounceResponse. Moreover, we set it statically to 60 seconds, so each peer will send a new AnnounceRequest passed 60 seconds to renew his actual state. However, that was at first, then we think about a more intelligent approach.

Let's think that we have a tinny file, of only some bytes of length, and we want to download it from a torrent. If we set an interval of 60 seconds, is highly possible that we download the file in 30 seconds and leave the torrent program or simply delete the torrent to stop seeding. In that case, the tracker master will be sharing my IP and Port for downloading that file for almost 30 seconds without knowing that I have finished the download. What we want to say is that the state of the download will be outdated after a period if we set a static value. Consequently, we should consider the number of peers at the swarm (if we have more peers, the download should be faster, so we talk about a seed/peer ratio) and the size of the file.

In our case, we finally implemented a selection mechanism considering the peers at the swarm and the amount of data that the peer must download. Actually, we don't know another fact that would be interesting to know: in the real torrent functionality, peers exchange information about other peers, so they can substitute the role of the tracker in this question. We can also consider the number of peers independently to consider the amount of time between each AnnounceRequest.

Peer selection mechanism

In the same way as the previous chapter, we started implementing a simple mechanism: simply return all the peers at the swarm. Obviously, the implementation was easy as a piece of cake, just return a collection with all the PeerInfos.

Checking the design briefly, we finally agree on the idea that we should check the state of each peer for each swarm. To tackle this issue, we classify peers in two groups: seeders and non-seeders. With this classification, we can send to the peer more seeders if is a leecher (is still downloading the file) or return a bigger amount of leechers if is a seeder (have downloaded all the file). Apart from that, we can return a different number of peers, depending on the integer value provided.

The implementation is more complicated than the previous one, due to the classification of peers. Due to that fact, the swarm's peer list is sorted by the amount of data left to download for each peer (ascending if the peer is a leecher and descending if is a seeder). Finally, we extract the number of peers that we need from that ordered list.

Validations and error handling

We do some validations at the program, mostly in the tracker side, when a new message is received in both directions (Peer-Tracker). Therefore, we will explain each one of those for every situation:

- **ConnectRequest** received at Tracker: we validate four cases.
  - **Size of the message**: if the size isn't equal to the ones expected, we send an error of SizeIncorrectError.
  - **ConnectionId**: if the id doesn't match with the assigned one or the default one for a first request (this one should be **41727101980**), we send an error of ConnectionIdIncorrectError.
  - **Action**: if the action isn't Connect, we send an error of ActionIncorrectError.
  - **Parse**: we try to parse the message. If the message parser returns null, we send back an error of ParsingError.
- **ConnectResponse** received at Peer: we validate four cases.
  - **Size of the message**: if the size isn't equal to the one expected, we retry the connection.
  - **Action**: if the action is Connect, then it's OK. If the action is Error, we retry the ConnectRequest.
  - **TransactionId**: we compare if the transactionId is the same that the peer sent before. Unless is, we retry the connection again.
  - **Parse**: we try to parse the message. If the message parser returns null, we retry the ConnectRequest.
- **AnnounceRequest** received at Tracker: we validate five cases.
  - **Size of the message:** if the size isn't equal to the ones expected, we send an error of SizeIncorrectError.
  - **TransactionId:** if the transactionId received doesn't match with any peer at the Tracker (this means that the tracker doesn't have that peer in memory, so it hadn't sent a ConnectionRequest before), we send an error of TransactionIdIncorrectError.
  - **ConnectionId**: if the id doesn't match with the assigned one before, we send an error of ConnectionIdIncorrectError.
  - **Action**: if the action isn't Announce, we send an error of ActionIncorrectError.
  - **Parse**: we try to parse the message. If the message parser returns null, we send back an error of ParsingError.
  - **Time interval**: we check if has passed the time specified since the last AnnounceRequest. Unless has passed, we send an IntervalIncorrectError.
- **AnnounceResponse** received at Peer:
  - **Size of the message**: if the size isn't equal to the one expected, we retry the AnnounceRequest.
  - **Transaction_id**: if the transactionId received doesn't match with one in the peer, we retry the AnnounceRequest.
- **ScrapeRequest** received at Tracker: we check two main things.

    - **Size of the message**: if the incoming message doesn't match any of the cases corresponding to the rest of messages (Connect, Announce), then we suppose that the incoming message is a ScrapeRequest.
    - **Parse**: we parse the message but if it returns null then we send an generic Error back to the peer stating that the message's size was incorrect.
- **ScrapeResponse** received at Peer: we validate four cases.
    - **Size of the message**: if the size isn't equal to the one expected, we skip it and wait for another ScrapeResponse.
    - **Parse**: the ScrapeResponse is parsed, if it returns *null* then we skip it and wait for another ScrapeResponse.
    - **Action**: we check whether the incoming message's action type is Scrape or not. If not, we skip it as always and we wait for another Response.
    - **Transaction_id**: if the *transactionId* received doesn't match the one in the peer, we skip the message and wait for another one. Instead, if the transaction is correct, then, we process the message and save the *ScrapeInfo* list in memory.