Departament d'Enginyeria Informàtica i Matemàtiques

GoTorrent – Gossip-Based BitTorrent

Task #1: **Asynchronous calls, gossip-based dissemination, push-pull gossip**

Distributed Systems 2016/17

You get so tired of having your work die.
I just wanted to make something that people would actually use.
Bram Cohen
[BitTorrent inventor]

The goal of this homework is to implement a prototype of a *BitTorrent*-like system by means of the Python *PyActor* library [1]. Although many parts of the standard *BitTorrent* protocol will be left as is, the dissemination of files will be via *gossip* communication protocols instead of the regular *BitTorrent* mechanism. A secondary, but not less important, goal of this assignment will be accordingly to learn the power of *gossip-style* communication. This style of communication can already be found in many modern distributed systems such as Amazon DB and Cassandra (Facebook), to name a few, and it is considered a basic building block for many large scale systems, here lies its importance.

1. BitTorrent

Here we provide a general overview of *BitTorrent* to help you understand the main concepts and the basic *BitTorrent's* ecosystem. The specific protocol to implement can be found in Section 3**.**

In BitTorrent, each file is split into *chunks*, and peers download not only from the peer with the complete copy of the file, called *seed*, but also download chunks from one another. This significantly diminishes the load on the seed.

Peers downloading a particular file are tracked by a *tracker*, a server that resolves queries and keeps track of which peers are downloading a particular file. All the peers downloading and sharing the same file, including the *seed*, form the *swarm* for that torrent file.

To join the swarm, a client must first download a metafile that contains the *tracker*'s URI, the file size of the torrent, the number of chunks, and another metadata. The client then connects to the *tracker* and requests a number of IP/port pairs of other peers to become member of the *swarm*.

The client then opens a number of connections to these peers, and start exchanging chunks among them. Once a chunk has been already downloaded, the client notifies each peer to which it is connected that it has completed the download of that chunk. Thus, clients are kept constantly informed of what chunks are available to them. As may be obvious from the above description, a file is not downloaded sequentially, i.e., each chunk of the file can be downloaded independently by a client.

2. Gossip

Gossip has been great of interest for large scale distributed systems for its appealing properties such as simplicity, robustness, and a lack of central control. The first real application of *gossip* was to make sure that each replica of the database on the Xerox internal network was up to date. Since then, *gossip-style* communication has been utilized in many large scale distributed systems.

The literature *gossip* on algorithms is extensive. In this assignment, we will focus on the simplest form of *gossip* for content dissemination. In its simplest incarnation, each node executes one process, referred to as the *active thread* in the literature. The *active thread* is run once in each Δ time units. We will call this waiting period a *gossip cycle*. At every gossip cycle, the active thread selects one or more random peers from the set of all nodes, and starts a communication operation. This operation can be a *push* exchange, a *pull* exchange, or a *push-pull* exchange, depending on the gossip-style adopted.

In *push* gossip, only the nodes that have a piece of information actively disseminate it within the population. In *pull* gossip, all nodes are active and query random selected peers for information about recently received messages, i.e., they do not passively wait for the reception of information pushed by other peers within the population. Finally, *push-pull* gossip combines both strategies at the same time. The pseudocode for a simple implementation of the *active thread* can be found in Algorithm 1.

```
Algorithm 1: Active thread
1:  loop
2:  wait($\Delta$)
3:  $p$ ← random peer
4:  if push and already has the update to disseminate then
5:      send the update to $p$
6:  end if
7:  if pull then
8:      send update-request to $p$
9:  end if
10: end loop
```

It is recommendable that the messages sent at the same time at every peer, that is, messages from the same *gossip cycle* do not mix with messages from other cycles. This assumption is not critical for the practical operation of gossip, but they are needed to better understand the pros and cons of each gossip strategy in this homework.

3. *GoTorrent*'s Protocol

Now that we have already introduced the main concepts in the previous section, we are ready to describe the specific protocol that *MUST* be implemented in this assignment. The protocol consists of two parts:

Part 1: Tracker Communication.

Like in *BitTorrent*, you must first implement a tracker which will be responsible of membership management functions through two methods: `announce` and `get_peers`.

Concretely, the `announce` method has two parameters, the `hash` or `id` of the torrent to be downloaded, and the reference to the peer that wants to participate in the swarm. For simplicity, you can use the file name as `id`. Note that, like in *BitTorrent*, the peer must periodically announce its presence in the swarm. The `tracker` removes peers from the swarm than do not announce themselves in a period of 10 seconds. The signature of this method is:

```
announce(torrent_hash, peer_ref)
```

The `get_peers` method is used to obtain a list of peers participating in this download. The `tracker` returns a fixed number of *random* peers from the swarm. The signature of this method is:

```
neighbors ← get_peers(torrent_hash)
```

Part 2: Gossip dissemination.

Instead of the conventional *BitTorrent* protocol, peers will use gossip to distribute the file chunks. To support it, you will have to adapt the implementation of the *active thread* listed in Algorithm 1 to disseminate the file chunks within the swarm. For this purpose, you can use the `intervals` functionality included in the *PyActor* library to allow an actor to periodically do an action.

Whatever the strategy will be, namely *push*, *pull* or *push-pull*, each node will select n peers at random from its peer set at the beginning of every gossip cycle. Recall that the peer set of a given peer is the set of all its neighbors that it got over time through the different calls to the method `get_peers` with the `tracker`.

Push gossip. In the simplest form, peers will only use *push* gossip. In this case the logic is very simple: for each peer p, the client will choose randomly a chunk and send it to p. To support *push* distribution, all the peers will have to implement the `push` method with the following signature:

```
push(chunk_id, chunk_data)
```

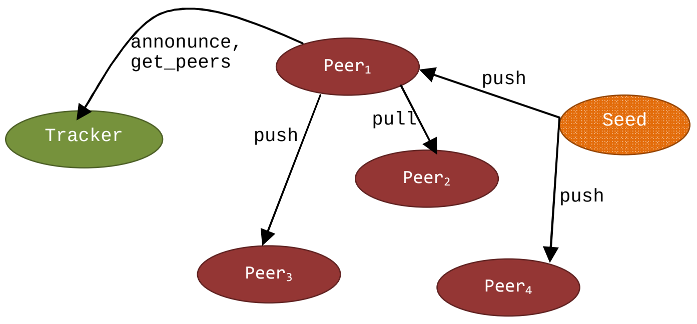
where the `chunk_id` is the identifier of the chunk and `chunk_data` is the actual data in the chunk.

Pull gossip. On the other hand, the *pull* strategy should work as follows. For each peer p, the client will simply request a missing chunk to p. This immediately raises the question of which missing chunk should be request by the *GoTorrent*'s client. In this assignment, the answer to this question has been left open. One simple approach would be to just request a random missing chunk. However, other strategies on piece selection could be applied. For instance, the client could check its neighbors in the peer set and request the chunk with the fewest copies, i.e., the *rarest chunk*. Irrespective of the adopted piece selection strategy, the signature of the `pull` method should be:

```
chunk_data ← push(chunk_id)
```

Note that depending on the piece selection strategy implemented, it may occur that the peer p has not the requested chunk yet. In that case, the `push` method should return `None` or launch an exception to signal that event.

In the next diagram, we provide an example of a *GoTorrent*'s swarm with 4 peers and 1 seed. The arcs are annotated with the corresponding method calls between the actors of the system. For sake of clarity, only the interactions for `Peer`$_1$ have been depicted in the diagram. Note that a *seed* will never pull chunks from the peers in the swarm, as it has already a complete copy of the file.



4. Requirements

The project will be implemented using *PyActor*. The specific requirements for this project are the following:

1. The deployment of the *GoTorrent's* system MUST consist of 1 tracker, 5 peers and 1 seed.
2. The length of a gossip cycle will be of $\Delta = 1$ second.
3. Initially, no peer will have partially or completely a copy of the file. Only the seed will have the file.
4. The implementation of the system MUST distribute at least the following piece of text: "GOTORRENT". Each character will correspond to a chunk. So the total number of chunks will be 9. That string MUST be mandatorily read from a file, so that the content of the file can be changed if the professor requires doing so. As the `chunk_id`, you can simply use the position of each character within the string.
5. After distributing the file, all the peers MUST have a complete copy of the file.
6. Every peer should `announce` itself to the tracker every `10` seconds. Before the distribution of the file is started, each peer, including the seed, MUST announce itself to the tracker.
7. A call to the `get_peers` method will return 3 random peers out of the 6 available in the swarm. This method will be called every 2 seconds by all the members of the swarm.

5. Tasks (Weights)

1. **(30%). Tracker.** Implementation of the tracker. Provide the appropriate tests to check that the tracker is working fine.
2. **(35%). Push gossip.** Implementation of *push* gossip. Also, to better understand the efficacy of *push* gossip, you should run the following experiment. Set the value of $n = 2$ and wait for the complete distribution of the string "GOTORRENT" in all the 5 peers in the swarm. During the experiment, and per gossip cycle, record the average fraction of the file that has been already downloaded by the peers in the swarm. Plot the results in a figure where in the x-axis there is the gossip cycle number and in the y-axis the average fraction of the file downloaded by peers. *Explain what you see in the figure. Do you believe that there are many redundant messages?*
3. **(25%). Pull gossip.** Implementation of *pull* gossip. Repeat the same experiment as described above, but in this case using your own implementation of the *pull* strategy. As before, and per gossip cycle, record the average fraction of the file that has been already downloaded by the peers. Plot the results in a figure where in the x-axis there is the gossip cycle number and in the y-axis the average fraction of the file downloaded by peers. *Explain what you see in the figure and compare it with that of push gossip. Which of both techniques is more efficient? Why?*
4. **(5%). Push-pull gossip.** Combine your implementations of *push* and *pull* gossip to implement *push-pull* gossip. Repeat the same experiment as described above and plot the results in a new figure. *Explain what you see in the figure. Is there much difference with the pull strategy alone?*
5. **(5%). Personal github.** All groups must create their own *github* repository with their code and documentation. If possible, we recommend including at least code coverage and health links. This requires the development of unitary tests for your code. Example of a https://github.com/danielBCN/PyActor-example

6. Deliverables

You will have to submit the **answers to all the questions**, the **final design document** as well as the code itself in any format (preferably **.pdf** and **.doc**); the allowed languages are Catalan, Spanish and English. The **final design document** should include a concise explanation of your design choices of about **3**-page length, implementation details, as well as what tests you have run to confirm that your implementation of *GoTorrent* works as specified. Do not forget to include **the source code** written in Python when submitting your solution.

Last available date: April 7, 2017;

 After the evaluation of the task, the professor responsible for your group may ask you about some points of your delivery to assess your degree of knowledge.

Final Hint: We recommend you to do not procrastinate and take advantage of the class hours in the laboratories.

Note: In case you find bugs in the code or the instructions, email *marc.sanchez@urv.cat* and I'll fix the issue, post revised versions on the website and inform everyone.

7. References

[1] PyActor library, 2017, https://github.com/pedrotgn/pyactor