LAB 10: DHT implementation with Flask (part 3)

RECAP (achievments after part 1 and part 2)

part 1: initial code setup

1. server.py –> defines the all web-methods constituting the API of a DHT node
   - /nodeinfo to get all known attributes of a running process
   - GET/PUT methods to get/update the predecessor or successor of a process
2. runserver.py –> aka "bootloader": script to deploy a node (i.e., it configures and starts the associated Flask app which is a web-server)
3. util.py –> defines common necessary functions, e.g.:
   - hashing function
   - clockwise distance
   - Process class to conveniently represent a process participating in the DHT

part 2: tasks, findNode, JOIN

We added major core functionalities to our DHT project, namely:

1. tasks.py –> exploits pyinvoke to
   - boot, destroy and reboot an initial network with two hardwired nodes
   - it exploits a locally defined *start_process* routine which manages the to spawn of new DHT processes properly linked to their logfiles
2. findNode –> a major DHT functionality!
   - embeds **iterative routing design**
   - exploits py-requests to serially get nodeinfo "successor-by-successor" (we called this mechanism *the LINEAR findNode*)
3. JOIN –> implemented first steps of this routine, namely:
   - get the responsible node that should welcome the joiner
   - issues *rewiring PUT requests*
   - still misses "task/client" that forges for us JOIN requests

Outline Today

1. Complete JOIN implementation
   - finish rewiring (with proper PUT requests)
   - new **add** @task to spawn a new process which asks to JOIN through some already existing process
   - new **addmany X** @task to rapidly grow our DHT
2. Persistent "NETWORK STATE", necessary to let asynchronously issued @tasks to know which processes are running

- *persist_process_list(processes)* –> dumps list of processes to JSON file
- *read_process_list(JSONfile)* –> to get info about running processes from previously written network state file

Where to use read/write NETWORK STATE?

- Spawning a process (read -> append 1 -> write)
- Overwrite network-state file @boot
- Remove it @killall
- Before JOIN read the network state, select a (random) node (the one that receives JOIN request from joiner), compute proper attributes for joiner. E.g.: don't use ports already bound to some process in *read_process_list...*
- **pick_random_process** –> very useful for all tasks, don't wait to implement this auxiliary method! :)

3. implement STORE(value)
   - server-side: **NB:** receiving a STORE request does not mean you must store the passed value!
   - rather: i) findNode, ii) then **insert** in that node (ergo, define an auxiliary **insert** Flask method)
   - client/tasks-side: new **store(value)** @task
   - why not a **storemany X** to rapidly issue many STORE(random value) requests and grow our DHT content
   - lipsum file to pick random content from it :)
4. implement LOOKUP(value)
   - server-side: as for STORE, separate the logic of handling a LOOKUP request from the fact of really GETting the looked-up value from the responsible node. **GET(key)** auxiliary method to get a content from the node that owns it
   - client-side: new **lookup(value)** @task that crafts a LOOKUP http-request

Bonus Track

1. **Monitor @task**
   - server-side: a method to get the content stored locally by a node
   - get via http requests the content of all running nodes
   - pretty-print it at regular interval to show evolution of the DHT
2. REGULAR UPDATE of the FINGER TABLE
   - Booting a node, use the Advanced Python Scheduler to start a background task which forces a node to update its finger table at regular intervals
   - Re-implement *findNode* using the now available fingerTable –> "auto-magically" your DHT will have become efficient :)
3. CONTENT-PASSING

You are having so much fun, aren't you?!? Then:

- Implement a very strange LEAVE... where a node does not leave when he wants, but when he is asked to! So, essentially, when you receive a LEAVE request, then you rewire your pred and succ together and then shutdown
- Before shutting down, pass your content to your predecessor

- Extend JOIN, get the content of predOfJoiner, determines which of those contents should me moved to Joiner
- Define a **delete(value)** method to delete from predOfJoiner those values that now you will store in Joiner