Middleware Labs: Chord

Petr Tůma Vojtěch Horký Antonín Steinhauser
Vladimír Matěna

April 17, 2018

Key-Based Routing (Chord)

- Download Chord implementation (python-chord)
  `http://d3s.mff.cuni.cz/teaching/middleware/files/python-chord.tar.xz`
  - Unpack anywhere, put symlink to ~/python-chord
- Get the files we prepared for you
  `http://d3s.mff.cuni.cz/teaching/middleware/files/as4.zip`
  - Simple chat application
  - Skeleton for the task implementation

Implementation

- Reuse as much code as possible
  - `chat.py`
  - `run.sh`
- Use already prepared data structures (dictionary, list, tuple)
- No deadlocks or race conditions
- Report issues, ask questions when unclear
  - Mailing list...

Notes

- Start the first instance
  - `./chat 127.0.0.1:5000 name1`
- Connect next instance running on the same machine
  - `./chat -j 127.0.0.1:5000 127.0.0.1:6000 name2`
- Documentation
  - On the following pages
    - Brief, but it shall be sufficient
  - Explore the prepared `chat.py`
    - Almost everything you will need for the task
    - `self.local_.user_command()`, `self.local_.register_command()`

API

- class Address - represents an address of a chord node (IP and port)
  - `Address.__init__(string_ip, int_port)`
  - `Address.__hash__()` id of the node with the address
  - `Address.__eq__()` address comparison
  - If all the nodes of the chord network are on the same host machine, then it is possible to use 127.0.0.1 as their IPs, however if nodes of the chord network are on multiple hosts, then 127.0.0.1 must not be used anywhere. Id of a node is given by hashing an IP and port and if a node has an IP 127.0.0.1 and another node from a different host knows it by an IP of its public network interface, it would think that the previous host has a different id than it really has (and it would break the chord network).

API

- Node - represents a chord node (shortcut for class Local or Remote), class Node does not contain all the methods listed below, but whenever we talk about an object of type Node in the following text, all the listed methods will be in it
  - `Node.id()` id of the node (given by its address)
  - `Node.__eq__()` node comparison
  - `Node.successor()` returns Node, returns the successor of the node. A successor is the node to which the information we keep should be migrated before we exit.
  - `Node.find_successor(id)` returns Node, for a given id returns the node that is responsible for it.
  - `Node.user_command(command, message)` on the given node invokes handler of the command (command must be a string without whitespaces), message is handed over to the command handler (message must be a string not containing new lines). The method returns string returned from the command handler or an empty string if there is no such handler on the node. (tip: use json.dumps to serialize parameters for the command and json.loads to deserialize them)

API

- `class Remote` - represents a chord node with which we might communicate through a socket
  - see Node above
  - `Remote.__init__(address)` - creates a stub for communicating with node with given address

API

- `class Local` - represents a local node of the chord network
  - see Node above
  - `Local.__init__(local_address, remote_address=None)` - creates a new local chord node. If `remote_address == None`, then a new chord network with one node is created. Otherwise, the local node will join an existing network through node given by `remote_address`. `local_address` is the address of the local node, it determines the local node's id.
  - `Local.start()` - starts the threads of the local node.
  - `Local.is_ours(id)` - returns whether the id belongs to the range our node is responsible for; useful for identifying ids that we are no longer responsible for.
  - `Local.shutdown()` - stops the local chord node (its threads)

API

- `class Local` - represents a local node of the chord network
  - `Local.register_command(command_name, handler)` - registers a handler for the command with name `command_name` (command name must be a string without whitespaces). The only parameter of the handler is a message passed to `user_command` method which caused invoke of the handler. The command handler must return a string value (returned value must be a string not containing new lines). The command handler may be invoked by a user_command method of a Node.
  - `Local.set_notify_handler(handler)` - sets the handler that is called, when the predecessor of the current node (and thus maybe the range of ids that our node is responsible for) changes. The new predecessor is passed to the handler as the only parameter.

API

- Be aware that the handlers registered by previous commands might be called from different threads at the same time.
- Following strings are forbidden as a command names, because they are used by the underlying chord implementation: `get_successor`, `get_predecessor`, `find_successor`, `closest_preceding_finger`, `notify`, `get_successors`

Submission

- **Python 3**
- By e-mail (deadline is on the web)
- Make sure it works in the lab downstairs
- The submission shall be easy to start