University of Oslo
Distributed Objects - Spring 2019
William Janoti (wcjanoti@ifi.uio.no)

Home Exam 1 Report

1 Source Files

- types.m - Type objects used throughout the application.
- server.m - Server logic, maintains file and peers indexes and provides search functionality
- peer.m - Peer logic, object that contains its own list of files and interacts with the server.
- hash.m - Hash algorithm implementation as a type object, the chosen implementation was DJB2 (`http://www.cse.yorku.ca/~oz/hash.html`).
- nopester.m - Main program, runs the test cases and simulates peer - server interaction.
- Makefile - used to compile and run the program, it has 5 targets: *compile*, *compile_planetlab* (using ec32), *clean*, *run* and *run_planetlab* (using emx32). The *run* target has an argument *host* to specify the master Emerald node to connect to. Example: *make run host=172.170.7*

2 Program Structure

2.1 Server

The server is defined in *server.m* and maintains two indexes, one associating the file's content hashes to a list of file names (as they might appear across the nodes) [Figure 1] and one to associate the file's content hashes to a list of peers where they exist. [Figure 2].

The server also keeps a list of current connected peers. The indexes were implemented using Emerald's *Directory*, for its simplicity for adding and removing items. Whenever a peer adds a new file, those indexes are checked and updated accordingly. Conversely, whenever a peer sends an update to let the server know that it no longer has a given file, those indexes are also update to remove references to that specific file.

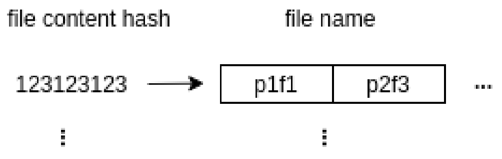

Figure 1: File names index

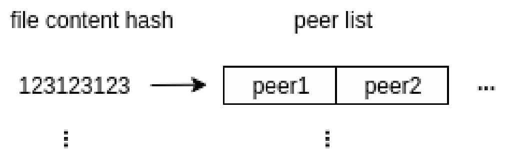

Figure 2: File peers index

2.2 Peer

The peer functionality is defined in *peer.m*, a peer can register a new file on the server, notify that it no longer has a file, update a file name and search/request a file from other peers. It maintains its own list of files in a Directory [3]

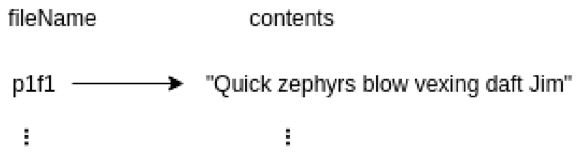

Figure 3: Peer file index

3 Testing

I chose to test the basic functionalities of the system as described in the assignment text, all test cases are implemented in *nopester.m*, except item 6 that was tested by manually killing a node and checking if the server got a notification. After each test case the state of the application is printed out.

1. Peer registering a file in the server
2. Peer searching and requesting a file available in only one peer
3. Peer searching and requesting a file that is available in multiple peers
4. Peer letting server know that it no longer has it
5. Peer updating a file (file name only)
6. Server notification of peer unavailable

The program expects 3 nodes to be available, one for the server and the other two to host the peers (5 in the test), there's a check for that in the code and the program will only run if that criteria is met.

For simplicity the file names follow the pattern *p + [peer number] + f + [file number]*, e.g. the first file created by peer 1 will have the name: *p1f1* and so on. The tests outputs are under the *test_ outputs* directory.

Item 1 is tested in the *setup* operation in *nopester.m*, it creates 5 peers and makes them register files on the server, also it is done so that there are a few overlapping files between the peers. This can be verified by checking the initial application state in *test_ outputs\initial_ global_ state.txt*

Item 2 is tested in the *testDownloadFileAvailableInSinglePeer* operation in *nopester.m.* In this case, Peer 1 (p1) downloads a file (p4f1) from Peer 4 (p4) and stores in under the name p4f1.downloaded. The state after this test can be seen on *test_ outputs\state_ after_ test_ case_ 1.txt*

Item 3 is tested in the *testDownloadFileAvailableInMultiplePeers* operation in *nopester.m.* In this case, Peer 1 (p1) downloads a file (p3f0) that is available in two peers (p2 and p3), the application should try to download from it the first available peer. The state after this test can be seen on *test_ outputs\state_ after_ test_ case_ 2.txt*

Item 4 is tested in the *testPeerRemovesFile* operation in *nopester.m.* In this case, Peer 4 (p4) removes a file (p4f2). The state after this test can be seen on *test_ outputs\state_ after_ test_ case_ 3.txt*

Item 5 is tested in the *testPeerUpdatesFile* operation in *nopester.m.* In this case, Peer 2 (p2) updates a file (p2f1) by changing its name from p2f1 to p2f1.updated. The state after this test can be seen on *test_ outputs\state_ after_ test_ case_ 4.txt*

Item 6 was tested manually. The relevant code is on *server.m*, lines 162 - 178.