Distributed Systems Assignment

pvxf29

04/03/2015

Using the code

Note: Code is in Python version 3.4.2
The code consists of three Python files, *client.py, fe_server.py* and *replica.py*. These files should be run concurrently in separate terminal windows. An instance of fe_server must be established first, and must stay running as it is the bridge between client and back end.

Each *client* must be run from a directory containing a database of valid frontend locations (fe*_server_db.p)*, which keeps a list of all the frontend servers that the client will attempt to query (allowing optional multiple front end querying). Upon starting, a client will print a list of commands with descriptions of their functionality.

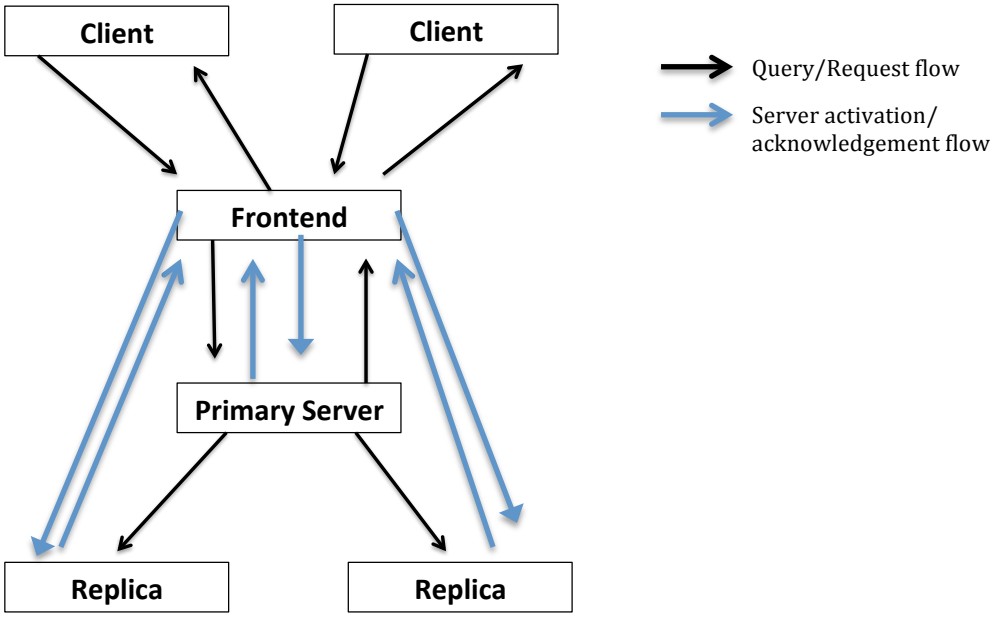
Each *fe_server.py* must be run from a directory containing *fe_port_db.py,* which contains the ports the frontend is allowed to create welcome ports at, and *active_server_db.py* which stores a list the frontend maintains of active backend servers.

Each backend server, *replica.py*, must be run from a directory containing an independent movie database (*movie_db.p),* and database of valid ports for it's own welcome ports *(be_server_db.p)* and frontend locations (*fe_server.db.p*). To create a three server system, three directories containing the above contents must be made, and each *replica.py* run.

All databases are implemented in the *pickle* Python module, which I opted for rather than simply including an array in the server file as it allows testing with differing database files, as well as maintaining and storing each database.

Implementation and Improvements

The system is structured as in the below diagram. The system consists of a frontend, which connects to any number (specifically 3 for the purposes of this project) of backend servers and can be contacted by any number of clients. The frontend receives requests from these clients and pushes those requests to a backend server allocated as a primary server. The primary server handles the request, pushing information from it's own database and the web back to the client, or propagating information changes to the backup servers (all other servers in the backend that are not currently the primary server).



Frontend – *fe_server.py*
The frontend server is a front end to multiple movie database backend servers, creating a transparent system and a simple experience for the client. When the frontend is started it finds a valid port to listen at and waits for a client or server to contact it.

When a server is created, it contacts the frontend to notify it that the server is active and available for client queries. The frontend stores this backend server's location in a database so that it knows which servers to contact later. The first server in this database is treated as the primary server. After storing the new backend server's location, the frontend responds telling that server it has been accepted.

When a client issues a request, the frontend attempts to forward that request on to a primary server. It tries each item in the *active_server_db.p* until it finds an active server which then becomes the primary server. If the frontend fails to connect to an item on the active server list, the item is removed as it is assumed to no longer be active. This ensures that if a primary server crashes a new primary can quickly be established, creating a dynamic system.

As the frontend handles each instance of client/server contact in a new thread, the system can take as many clients as wish to connect to the system.

Backend Server - *replica.py*
As previously stated, when a backend server is created it attempts to find a frontend server from it's list of valid frontend locations. If it finds a frontend, the backend server sends it's location and awaits confirmation of it's acceptance by the frontend. If no frontend is found then the server closes as it cannot be queried by the system.

Once it has been accepted by a frontend it await contact from the frontend containing a client request. If the request is to retrieve information about a movie, the server queries it's database and looks for an entry of the queried name. If no existing entry is

found, it then looks online for the movie utilizing OMDb API, stores this retrieved data on it's database and any backup server databases, and then returns the information to the client.

If a movie is to be edited, the server checks that the entry already exists and updates the URL and Description fields. If the entry doesn't exist already or a client wishes to add a new movie to the system, the server checks that and entry of the same name doesn't already exist and add the movie. After updating it's own entry, it pushes the update to all backup servers.

The primary backend server is the first active server on the server list maintained by the frontend. When the primary server wants to propagate information to the rest of the backend it requests the server list from the frontend and requests and information update to each other server in turn, ignoring itself. The backup servers operate in exactly the same way as the primary server, but will not propagate information to other servers as they check they are slave servers (rather than primary servers) after checking the active server list.

The information retrieval from the system's own database requires exact string matching, whereas the retrieval from OMDb returns partial matches, which results in an inconsistent user experience. For example, searching for 'The Ring' on OMDb returns 'The Lord of the Rings: Fellowship of the Ring', whereas searching for 'The Ring' in the database would return The Ring (assuming an entry for the movie existed). Ensuring consistency of search methodology would improve the system.

Client – *client.py*
The client system takes requests from the user via *input()* and attempts to push those requests to a frontend, which it attempts to find at one of an array of locations it has stored. The frontend handles the request as previously stated, and then returns the correct response. Because the connection to the front end is established as the client attempts to query, the frontend crashing and restarting at a different address is not an issue.

Test Cases

- Primary server goes offline before client command is relayed to it -> Frontend removes primary server from active server list and allocates next server as new primary serve.
- Primary server goes offline whilst handling client request -> Frontend try containing request to server reaches exception; Frontend removes primary server from active server list and allocates next server as new primary server.
- Client requests information on a movie that is not in the database -> OMDb queried for closest match and information retrieved.
- Client requests information on a movie where database has incomplete record of movie information -> Missing fields retrieved from OMDb.
- Client goes offline after sending request -> Servers complete the process and reach exception when sending information back to client. Go back to listening state.