Design Doc for a Load Balancer

Shahar Dubiner,
sdubiner@ucsc.edu

Objective

Implement a load-balancer that redirects client requests to one of several servers based on some measure of the servers' health.
It is assumed all the servers can respond to health check queries, in a manner similar to that supported by the multithreaded logging servers from assignment 2.
Usage:

```
./loadbalancer <port_num> <server_port_num [server_port_num]*> \
                    [-N <max_parallel_connections>] [-R <rate_healthcheck_per_num_requests>]
```

The input parameters may appear in any order, including mixing port_numbers between the flag arguments.
For example: ./loadbalancer 8080 -N 8 1234 -R 7 5678 1248
load-balancer's port is 8080 and there are 3 servers with ports 1234, 5678, and 1248. The maximal number of parallel connections handled is 8. Health-checks of the servers are performed at least once per 7 requests.

Load Balancing

It is assumed all servers have access to the same resources (file system). It would make sense to keep track of how many requests are currently handled by the server, what is the average response delay for the last few requests etc. However, the assignment specifies precisely how servers are to be compared:

```
bool compare(server1, server2) {
        if (server1.up && server2.down) return true;
        if (server1.down && server2.up) return false;
        if (server1.num_req == server2.num_req) return server1.num_err < server2.num_err;
        return server1.num_req < server2.num_req;
}
```

Where num_req and num_err are the statistics the server reported in the last successful health-check and the up/down status of a server is based on whether it has reached timeout for some query and its response to the last health-check query. If the status in a health-check response is not 200 (OK) or its format is incorrect in any way (or it has reached timeout), the server is marked as being unhealthy (down) until the next successful health-check. A server that is down is marked up again if it responds properly to a health-check request.

Therefore, it makes sense to keep (at least) the following information for each server:
char* port, int num_err, int num_req, bool up.

Matching Clients and Servers

Once the load balancer matches a client with a server, it needs to remember the match, so that any buffer arriving from a client/server socket-id is forwarded correctly to the corresponding server/client socket-id. Moreover, each such match needs to include also a timestamp and the server itself. This way, if there is no connection for an extended period of time, I can mark the server as being down and send a 500-error message to the client.
The natural solution for this problem is to have some kind of mapping between socket-ids
M: id1 → {id2, struct serverInfo*, timeval}
Initially, I intended to implement a simple hashmap holding lists of key-value pairs based on key%HASH_SIZE, but then I realized the socket-ids I can handle are restricted by FD_SETSIZE anyway, as I am using select(). Furthermore, I am expected to handle at most N requests concurrently, so a simple list of up to N elements or an array of size FD_SETSIZE, with access time O(1) would work as well. However, since I've already implemented the hash map and passed all the tests, I left it as is.

Health-Checks

The load balancer sends periodic health check requests to all the servers. These requests should be treated in a similar way to regular clients' requests, so as not to block the flow of the load balancer while waiting for a response. The socket-ids for such requests all "match" with the socket id of the load balancer. In order to keep track of these health check requests, I intend to add a socket-id and request_time field for each server. Thus, the data kept for each server would have the following structure:

```
struct serverInfo {
        char* port;       // the server's port number
        int num_err;      // number of errors based on the latest health check response
        int num_req;      // number of requests based on the latest health check response
        bool up;          // indicates whether the server is up (true) or down (false)
        int hc_id;        // health check request id
        struct timeval req_time;  // time request was sent
};
```

When the load balancer reads data from a socket-id, it needs to follow these steps:

- If the socket-id is of a health-check request:
    - Get the health-check data (including up status) and update the server info.
- If not:
    - Forward the data received to the matching socket-id.

When the load balancer checks socket ids for timeout events it checks the last_communication field for client-server pairs and the req_time for health-check requests.

Tuning

I have some control on parameters that are to be hard-coded and fixed:

- max_timeout: maximal timeout beyond which a server is considered 'down'. It makes sense to have this value depend on the buffers used by the servers. I do not want this value to be too low, because busy servers might take time to respond to queries already assigned to them. On the other hand, I want to be able to identify when a server is down as early as possible, so as not to send it more requests that would eventually receive a 500 (SERVER ERROR) response. I intend to start with something like 3 sec, and see how it performs.

- health check frequency: I need to perform health checks on all the servers every R queries (which is an optional input parameter), or every X seconds, whichever happens first. I do not want to have X very small, because then the health-checks would be a noticeable interference. On the other hand, the health-checks provide information relevant to the current status of the servers. In particular, I can conclude whether servers are up or down (in addition to the number of errors and queries). If I know a server is down, I could avoid directing client requests to it, and the overall quality of the load balancer would increase. Thus, the smallest X that would not overwhelm the system would be ideal. I intend to start with 1, and experiment with it while testing my load-balancer.

The basic guideline when setting these two parameters is: make max_timeout as large as possible and the health-check frequency as small as possible, as long as the performance of the load-balancer is not affected negatively.

Handling Multiple Requests and Responses

The accept(), send(), recv() methods all work sequentially and can block the flow of the program. Two ways to avoid having them become bottle-necks are suggested in the spec:

1. Use non-blocking sockets.
2. Use select().

I do not know which would work better or would be more convenient to use. However, since we were provided the sample code that uses select(), and since I see this as an opportunity to get familiar with this new method, I intend to use select(). My current understanding is that the non-blocking sockets approach would basically involve calling fctrl() after creating the socket. Using select() does limit the number of open socket-ids and maximal socket-id I can use.

Flow

After initial settings, the load-balancer goes into an infinite loop (exited only when errors occur or control-C is pressed).
Within the loop:

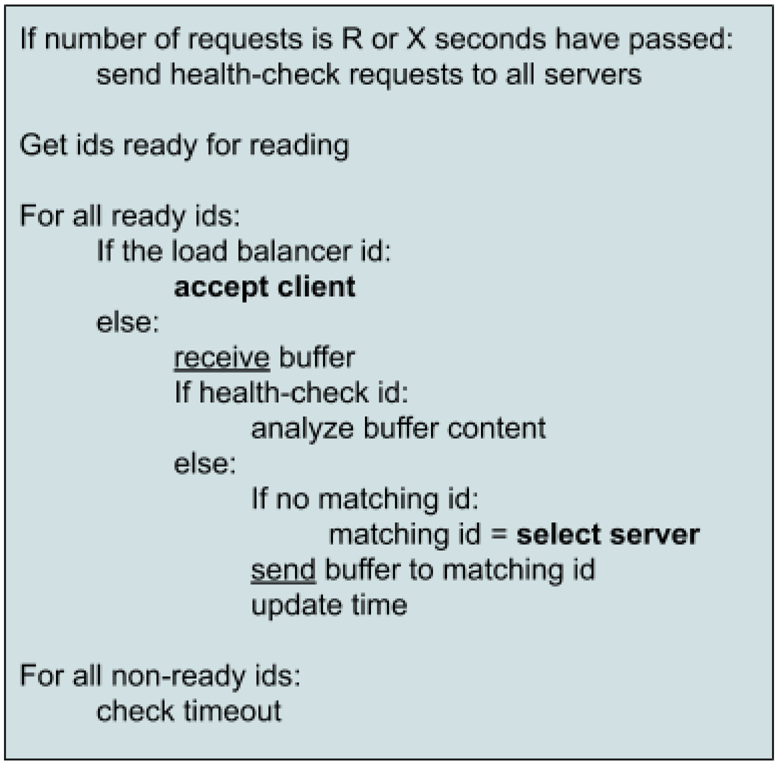
1. Perform a health-check for each server, provided the R-condition (R requests sent since last check) or X-condition (X seconds have passed since last check) are satisfied.
2. Use select() to see which of the socket ids has something happening for it.

For each ready id, exactly one of the following cases would hold:

3. The id is of the load balancer, meaning a new client request has arrived: create a new id (using accept()) for this client and increment the number of requests.
4. The id is of a health-check request:
    a. receive the buffer (using recv()). The response is ready and short, so make sure to read all the bytes.
    b. analyze the response.
5. The id is not the load balancer or of a health-check and does not have a matched-id: select the best server (defining a new client-server pair), or return 500-err if there are no servers.
   ⇒ For the process of selecting the best server, see the 'Load Balancing' section above.
   ⇒ Once there is a matched-id, continue to step 6 below.
6. The id has a matched id. Read the bytes to a buffer (using recv()). This will not block, because the id is ready. Send the buffer to the matched-id (using send()).
   update the time in both Map entries (of the id and the matching id).

For each of the ids that are not ready:

7. Check timeouts by comparing the current time to the time kept for the id and its pair or in the serverInfo structure, if the id is of a health-check.
    a. If a client id has been waiting for too long, send error-500.
    b. If a pending health-check is taking too long, mark the servers as being down.



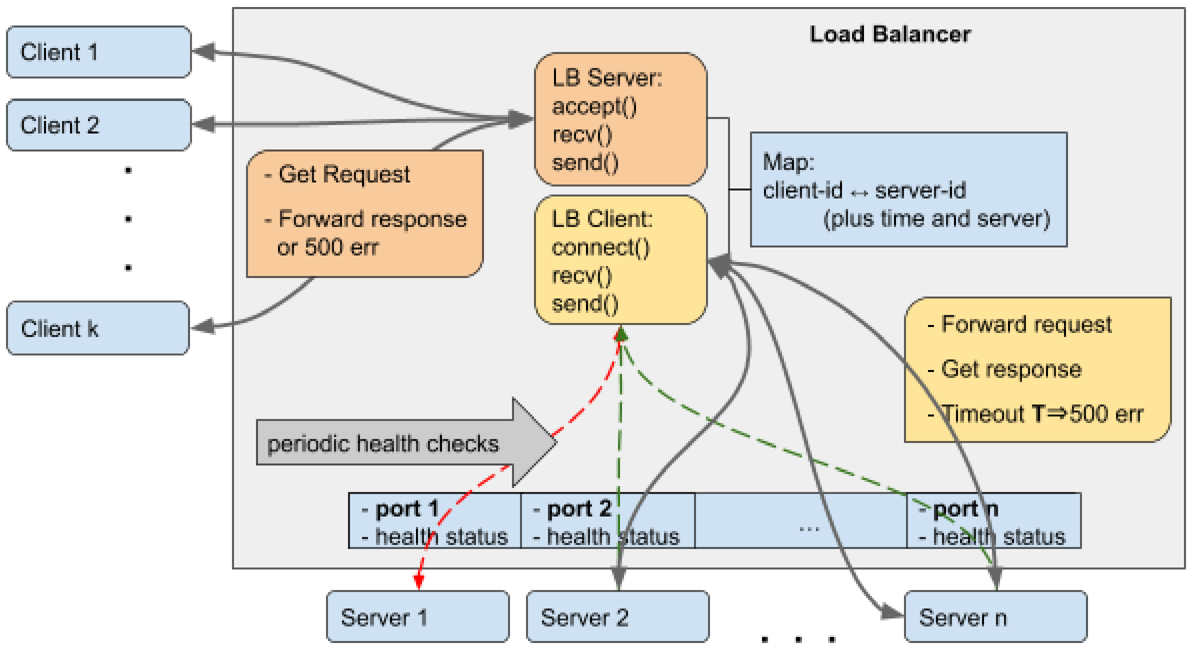
```
If number of requests is R or X seconds have passed:
    send health-check requests to all servers

Get ids ready for reading

For all ready ids:
    If the load balancer id:
        accept client
    else:
        receive buffer
        If health-check id:
            analyze buffer content
        else:
            If no matching id:
                matching id = select server
            send buffer to matching id
            update time

For all non-ready ids:
    check timeout
```

The diagram below shows the data flow between the clients, load-balancer, and servers.


Load Balancer
Client 1
Client 2
Client k
LB Server: accept() recv() send()
- Get Request
- Forward response or 500 err
LB Client: connect() recv() send()
Map: client-id ↔ server-id (plus time and server)
- Forward request
- Get response
- Timeout T⇒500 err
periodic health checks
- port 1 - health status
- port 2 - health status
...
- port n - health status
Server 1
Server 2
Server n