

CSIS0234B Computer and Communication Networks (Class B)

Tutorial 2

Extremely busy servers

You are given an iterative server, and a client program to test the efficiency of the server. Your task is to test the efficiency of the server when different methods of concurrency is used.

Like any benchmarking, it is essential that we document what we are testing, so that others can repeat it. The protocol is very simple: the server sends a character ('h', "hello") to the client and waits until the client makes a one-character reply. The client will do the reverse, read a character and write another back to the server. You can try them by running `it-server` and then `client <hostname>`. If you use the same computer to act as a client and a server, you can use `localhost` as the hostname. But if possible, try to test with two computers as well.

1. The iterative server

The current server has the following at the end to act as an iterative server:

```
int listen_sockfd = get_listen_socket();
if (listen(listen_sockfd, 5) == -1)
    err(1, "listen");
/* Iterative server */
for (;;) {
    int connected_socket = accept(listen_sockfd, 0, 0);
    do_serv(connected_socket);
    close(connected_socket);
}
```

The `do_serv()` function performs all the interactions between the server and the client, and you should not need to touch it at all. All your work should be on the main function (and possibly add some header files to the program).

Occasionally the ports of a computer will be too busy, and you might want to change to use a different port. There is one option for the server, `-p port_num`, which allows you to run it over a different port. By default the server runs on port 7777. E.g., to run it on port 7778 instead, use `it-server -p 7778`.

2. The client

The client does the simple thing to try completing as many connections to the server as possible. It uses `fork()` to create a few processes, all of them trying to connect to the server and perform the interaction repeatedly. The parent process then waits for some time to pass, and then ask all children to stop. On stop, each child prints how many successfully and failed connections are made. You shouldn't need to change the code, so it is not shown here. There are a few options to tune the client:

1. `-c num_children`: number of children to create, by default 10.
2. `-t test_time`: amount of time to test the server, by default 5 seconds.
3. `-d conn_delay`: amount of time to wait during a connection, by default 0ms. The Linux timer is of granularity of 10ms, so specifying 1 is the same as specifying 10.

4. `-p port_num`: port number, by default 7777.
5. `remote_host`: the host name of the server to test.

3. Your task

1. Test the iterative server. Use different values of `conn_delay` (e.g., 10 and 100) to see its effect.
2. Change it to a concurrent server, and test it again. Compare the results with that in step 1.
3. Change it to a pre-forking server, and test it again. Compare the results with that in step 1 and 2.

4. Hints

1. Work in groups of 2 to 3 students.
2. To make it easy for us to prepare the computers for the next tutorial group, the files `client.c`, `client`, `it-server.c` and `it-server` are write protected. Copy `it-server.c` to another file before making modifications to it.
3. After each test, wait for at least 1 minute before starting another. This is to avoid connection errors due to port numbers being used up. You can use `netstat -t` to see whether any old connection is still in `TIME_WAIT` state (waiting for some time to prevent old connections to interfere with new ones).
4. Since TCP allows retransmission, connection will not fail because of insufficient backlog. (See man page of `listen()`). The failures of the client you see are due to the problem above (i.e., port number cannot be used yet).
5. Our test will make some thousand connections (or even more). If you are not careful, many OS limits can be exceeded. So take the following precautions:
 - Make sure that the connected port is closed at the end of each server cycle. Otherwise, the server will soon get an error of “too many file descriptors” whenever `accept()` tries to make a connection.
 - Make sure that the child processes properly `exit()` the program rather than going back to the path of the parent. Otherwise all the children will at the end try to run as the server, soon making the computer very slow.
 - For the concurrent server: When a parent `fork()` a child, the OS normally expects you to wait for it, e.g., using `wait(0)`. When the child dies, the OS will not collect the process number for reuse until you do so. Such a “dying process” is called a zombie process. You must prevent zombie processes from being accumulated. Otherwise the computer can run out of process numbers. This means you must wait for the child unless the parent dies. Alternatively use `sigaction()` to set the signal action of `SIGCHLD` to `SIG_IGN` to inform the system that you are not interested in the children.