# Realizing Servers for Multiple Clients

**Course:** Computer Network Lab (AV 341) **Experiment-5** **Date:** 26th February, 2018

**Description:** The experiment deals with designing a server process that can handle multiple clients simultaneously using TCP. The basic idea is to create a socket for listening connection requests and create separate sockets for communication upon accepting connection requests. However, for serving multiple clients (i.e., managing multiple connection sockets) at the same time, we can use two approaches: (*i*) iterative server and (*ii*) concurrent server. In addition to realizing the above-mentioned approaches, we need to analyze the Round-Trip-Times (RTTs) incurred in three types of servers.

## 1 Iterative Server

In iterative approach, the clients are served in a round-robin fashion, i.e., each client is assigned with a time slot from the server in circular order. For example, consider a server handling three clients $C_1$, $C_2$, and $C_3$. First, server processes (here processing means echoing the received packet back to the client) data from $C_1$, then from $C_2$, and finally from $C_3$. Then server comes back to $C_1$ and process continues (Figure 1).

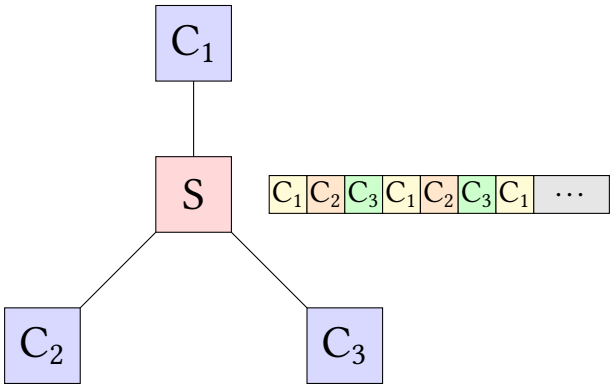


**Figure 1:** Scheduling of clients in iterative server process.

One limitation of iterative approach is that, *before starting the communication, all clients need to establish connection with server.* That is, the number of clients should be known beforehand to set up an iterative server.

## 2 Concurrent Server

Even though iterative server handles multiple clients, the processing follows a sequential order. Also, a client is allowed to communicate only during its respective time-slot. For example, $C_3$

needs to wait until the server completes communication with $C_2$. In order to overcome the drawbacks of iterative approach, concurrent servers are proposed. In concurrent server, a separate execution sequence (either as a new process or a thread) is created for serving each client. Therefore, clients can connect and communicate to server at any time. Concurrent servers are realized using two programming constructs: (*i*) multiprocessing and (*ii*) multithreading.

## 2.1 Using Multiprocessing

Here, server creates a separate process at the server for each client upon accepting a connection. In C, the *fork()* system call is used to create a new process (called the *child process*) from a process (called the *parent process*). The child process includes the code for serving the client. An example for creating a child process is given in the following code:

```
#include<sys/types.h>
#include<stdio.h>

int main()
{
  pid_t pid;

  // Creation of a new process
  pid = fork();

  if(pid)
    {
      printf("I am the parent %d with child %d.\n", getpid(), pid);
      // Statements for parent process
    }
  else
    {
      printf("I am the child %d\n", getpid());
      // Statements for child process
    }

  return 0;
}
```

## 2.2 Using Multithreading

Thread is an independent execution sequence *within a process*. Thread is also knows as *instance of a process* or a *light-weight process*. Unlike multiprocessing, where separate processes possess their own resources, threads in a process share the same resources as that of the parent process. In our case, thread defines the code segment for serving the client and new thread is created for each client. An example for a process with two threads is shown in the following code:

```
#include<stdio.h>
#include<pthread.h>

// Array to store thread ids
pthread_t tids[2];
```

```c
// Global variable which can be shared among threads
int j = 0;

// Function definition for thread-1
void* fun1(void *args)
{
  int delay;
  delay = *(int*)args;

  printf("Thread with fun1 created with id: %lu\n",\
         pthread_self());

  while(1)
    {
      printf("fun1: j = %d\n", j);
      j++;
      sleep(delay);
    }
}

// Function definition for thread-2
void* fun2(void *args)
{
  int delay;
  delay = *(int*)args;

  printf("Thread with fun2 created with id: %lu\n",\
         pthread_self());

  while(1)
    {
      printf("fun2: j = %d\n", j);
      j++;
      sleep(delay);
    }
}

int main()
{
  int status, interval[2] = {3, 5};

  // Create and execute thread 1
  status = pthread_create(&tids[0], NULL, &fun1, (void*)&interval[0]);
  if(status)
    {
      printf("Thread with fun1 creation failed.\n");
    }

  // Create and execute thread 2
  status = pthread_create(&tids[1], NULL, &fun2, (void*)&interval[1]);
  if(status)
    {
      printf("Thread with fun2 creation failed.\n");
    }
```

```
    // Waiting for threads to complete
    pthread_join(tids[0], NULL);
    pthread_join(tids[1], NULL);

    return 0;
}
```

**Note:** Programs using threads need to be compiled with an additional option *-lpthread*. For example, server program with multithreading can be compiled as

```
$ gcc server.c -o server -lpthread
```

Our experiment can be done in three phases:

**Phase-1** Iterative server

**Phase-2** Concurrent server with multiprocessing

**Phase-3** Concurrent server with multithreading

Finally, we need to compute and tabulate the Round-Trip-Time (RTT) (averaged over $N$ messages) in three types of servers. As an example, you can create a table as follows:

**Table 1:** Comparison of RTTs incurred in iterative and concurrent servers.

| **Server Type** | **RTT** |
|---|---|
| Iterative | ... |
| Multiprocessing | ... |
| Multithreading | ... |