Assignment 2 - Distributed Counting

CIS 570, Fall 2013

Due Date: October 29, 2013 11:59pm

1 Introduction

The object of this assignment is to familiarize yourself with several key concepts in the area of distributed systems. The first is concurrency control for shared resources. The second is communication between nodes in a network, in this case a server and several clients.

For this project, you will be responsible for the creation of a client and a server. The problem we are solving in this homework is one called distributed counting. Our solution will entail a server that has a single counter variable. Clients will connect with the server and request the counter be updated using simple add and sub commands. Thus, through their interaction with the server, clients that are independent will be able to maintain a single global counter.

2 Specification

There will be three distinct parts of this project, the client, the server and a tester. An outline of each is described below.

2.1 Client

The client will be a simple command line interface that will allow a user to connect to the server and access the server’s counter variable. Upon startup, the client will immediately establish a connection with the server (detailed below). The client has two modes of execution, an interactive and an auto mode. In the interactive mode, the client will provide a command line prompt for the user to send commands to the server. The valid commands are detailed below. In addition, the result of each command will be printed to the terminal. A sample output of this is below.

./client interactive
Connecting to server at 127.0.0.1:7000...success!
client--> add 2
current counter value: 6
client--> sub 2
current counter value: 4
client--> exit
Goodbye.

In the auto mode, the client will connect to the server and automatically request the counter be incremented by 1000 by requesting the counter be incremented by 1 1000 times. Thus, the client will send 1000 individual messages to the server and receive 1000 replies. The client should NOT disconnect/reconnect to the server after each command; all 1000 commands will be sent using the same initial connection. In auto mode, no input is required from the user. A sample output of this is below:

./client auto
Connecting to server at 127.0.0.1:7000...success!
initial counter value: 0
Sending add commands...done!
final counter value: 1000
Goodbye.

Note, it may be possible that the initial and final value for counter will not be exactly 1000 apart if other clients are simultaneously sending add/sub commands.

2.2 Server

The server will contain a single global count variable named counter. It is this variable that will be incremented/decremented upon requests from clients. The initial value of counter should be 0 and should be initialized when the server is started.

The server will listen for connections on port 7000. Upon receiving a connection request from a client, the server will accept the connection request and spawn a new thread using the pthread API. All updating to counter should be done in the spawned threads, not the server thread. This means that after a new thread is spawned, the server thread will immediately go back to listening for incoming connection requests from other clients.

Because it is possible to have multiple concurrent client connections that all share the counter variable, it will be necessary to use concurrency control mechanisms to ensure mutual exclusion. This should be done with the use of a single mutex from the pthread API. Any section of code accessing the counter variable will be considered a critical section and will be protected using the mutex.
2.3 Tester

The purpose of the tester is to simulate concurrent access to the server by multiple client threads. The tester will take in a single command line argument that specifies the number of clients to start. Each client will be started as its own process and in the auto mode described above. This can be done using the fork() and execvp system calls you used in Assignment 1. Tester must wait for all client processes to finish execution before exiting. Because the clients are started in auto mode, each will connect with the server and increment the counter variable by 1 1000 times. For example, calling ./tester 5 will result in counter being incremented by 5000.

2.4 Client-Server Communication

The client and the client will communicate using the Berkeley socket API discussed in class. For simplicity, both the client(s) and the server will reside on the same machine, i.e. localhost, or IP address 127.0.0.1. Communication will be done using port 7000, meaning that the server will bind/listen on port 7000 and the client will connect on port 7000. All communication will be connection-based and will be done with the SOCK_STREAM option for both the client and the server sockets, do not use SOCK_DGRAM.

Once a connection between a client and server has been established, the two can now communicate using messages. For this project, the API between the client and the server will be simple addition and subtraction requests. These commands will be sent from the client to the server. These are detailed below:

- **add <unsigned int num>**
  
  Increments counter by the amount specified in num. As a response, the server will reply with the current value of counter after the specified amount has been added.

- **sub <unsigned int num>**
  
  Decrements counter by the amount specified in num. As a response, the server will reply with the current value of counter after the specified amount has been subtracted.

- **get**
  
  A request from the client to get the current value of the counter variable. No change to counter is made.
3 Submission

Your program should compile and run on the Linux workstations available in the CS lab (Lab DI0N 305). You will submit 3 source files for this project: `client.cpp`, `server.cpp`, `tester.cpp`. Include a heading at the very top of each file including the assignment number, your name, a description of the code contained in this file, and any instructions for compilation and execution. Your code also needs to be well-documented, with any major constructs (i.e. functions) clearly commented. In addition, you will submit a Makefile that will compile your code.

Any code that fails to compile will receive a 50% penalty in addition to other possible penalties, so make sure you test it! As per the course policy, late submissions will receive a 10% deduction per 24 hour period they are late. All work is to be done independently and must be your own. Any collaboration, even in the form of discussing solutions, will result in 0 for this assignment. The grading for this assignment will follow the following rubric:

- Documentation: 20%
- Overall Design: 20%
- Functionality: 60%

All files need to be submitted as a single zipped file through the MyCourses site. Please submit only one copy of your program. If multiple submissions are made, only the latest copy will be graded, no exceptions.