Sheffield Hallam University
College of Business, Technology and Engineering
BSc Computer Science (Year 3)
55-600738: Concurrent and Parallel Systems

Module Leader: Dr Sergio Davies		Level: 6
Module Name: Concurrent and Parallel Systems		Module Code: 55-600738
Assignment Title: Discussion Forum		
Individual	Weighting: 60%	Magnitude: *2 Pages A4 (excluding graphs and tables)*
Submission date/time: Thursday, 8 December 2022, 3 pm	Blackboard submission and Turnitin submission	Format: Word, executables and source code (see below).
Planned feedback date: Thursday, 5 January 2023	Mode of feedback: Blackboard	In-module retrieval available: Yes

Module Learning Outcomes

- LO1. Demonstrate understanding of the principles of both parallel and concurrent architectures and the importance of such architectures across a range of computing applications.
- LO2. Analyse problems to discover their inherent parallelism or concurrency and use appropriate tools and techniques to develop and implement solutions.
- LO3. Analyse parallel or concurrent algorithms to understand their performance and look for common problems such as livelock, deadlock or resource starvation.
- LO4. Implement software which demonstrates the principles of both concurrency and parallelism.
- LO5. Critically evaluate concurrent and parallel technologies and applications and produce informed judgements about them.

Assessment Specification (2022/23)

Your task is to implement a C++ server for a simple discussion forum geared towards high traffic and short messages. The server should allow multiple clients to connect concurrently over the network, and to post and read messages on multiple topics.
You must test your server in high-load scenarios and evaluate its throughput, depending on the number of connected clients and the nature of their interaction.
You must submit C++ code for the server, your test and evaluation harness, and you must submit a short report on your throughput evaluation, details follow.

Technical Specification

Server

The server will communicate with clients over TCP sockets, listening for connections on port 12345. Server and clients exchange requests and responses following a simple line-based protocol that is specified in detail in the file “message board protocol.html” which is available on Blackboard.
To support your development efforts, on the blackboard you can find:

- A C++ parser for the request format, with a “main” function showcasing how to use it.
- TCP server and client libraries to provide you with basic network communication facilities between the server and the client. Each Visual C++ solution implements a “main” function. If they are run on two different command lines (first the server, then the client), they are configured so that the client asks the user for a string to be sent to the server. The server reverts the string and sends it back to the client.
- A protocol verifier, which connects to your TCP server and verifies that the communication follows the required protocol.
- A reference implementation, which will provide the basis for the comparison of the speed of your implementation.

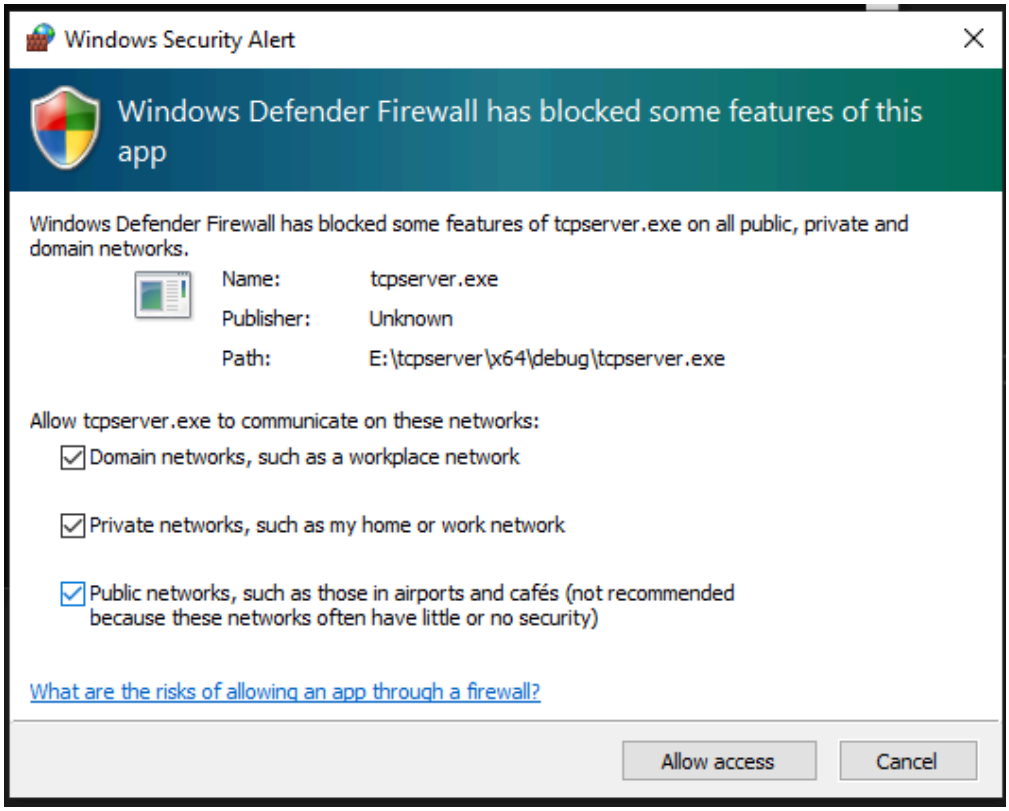
Despite the parser and the TCP server and client being provided, optimizations in the code of these modules are allowed and encouraged, provided that all interfaces are kept equal in the new version.

Compiling and running the TCP server and client

The TCP server can be run from the Visual Studio interface or from the command line (after compiling the code) with the command:

```
tcpserver.exe
```

The first time the server is started, Windows may ask if your process is allowed to accept incoming network connections using a request window similar to the one in the image below (some portions of the text may change, depending on your specific setup):



To allow incoming network connections, it is important to tick all the three checkboxes and press the button “Allow access”.

The TCP client requires a parameter on the command line to identify the PC on which the server is running. Therefore, it needs to be compiled through the Visual C++ interface, and then run via command line. If you are running both the server and the client on the same PC, you can use the loopback interface, starting the TCP client with the command line:

```
tcpclient.exe 127.0.0.1
```

Alternatively, if you are running the server and the client on two different PCs, the client can be started by identifying as a parameter on the command line the IP address of the computer where the server is running:

```
tcpclient.exe x.x.x.x
```

Where the string "`x.x.x.x`" identifies the IP address mentioned above. When started, the client will request a text string to send to the server. Entering an empty string results in the termination of both the client and the server.

Protocol verifier

The protocol verifier is a simple executable which tests the communication protocol with the server. It behaves like a TCP client executable, and the command line to start it is:

```
ProtocolVerifier.exe 127.0.0.1
```

Alternatively, if you are running the server and the verifier on two different PCs, the protocol verifier can be started by identifying as a parameter on the command line the IP address of the computer where the server is running:

```
ProtocolVerifier.exe x.x.x.x
```

Where the string "`x.x.x.x`" identifies the IP address mentioned above.

When the protocol verifier runs, it sends a known set of strings to the server, checking that the replies correspond to the specifications. At the end, the protocol verifier indicates the correct replies from the server and the non-compliances.

Throughput test harness

The test harness should fire up *m* poster threads and *n* reader threads, each connecting to the server. A poster thread posts messages (at a high rate) on a number of topics. A reader thread reads messages (at a high rate) about a number of topics. Each thread logs how many requests it manages to complete in a set unit of time (e.g. 1 second). The numbers *m* and *n* must be configurable.

Throughput experiments

Vary the number of poster and reader threads between 1 and 3 and evaluate how many POST and READ requests per second the server manages. You should repeat each experiment several (typically 10) times and report average throughput rates and the observed variance.
You should write a report (1 or 2 pages of A4, excluding tables and graphs) describing your experiments, with a particular focus on the strategies for generating post and read requests. Report all your data as a table, and present averages as graphs. Reflect on what the figures tell you about your server.
You MUST perform these experiments on the lab machines in Cantor 9341 for replicability. Record the IP address of the machine you are using.

Assessment Criteria

Your server will be assessed mainly based on its speed. You will need to optimise your code using the multithreading principles and techniques learned during the course and analysing the execution speed of your code to identify the functions that require the longest (using the visual studio profiling tool) and trying to optimise them.
The assessment will be mainly based on comparing the execution speed of your submitted code against a standardised test harness (not your own test harness) that is shared with you as the “reference implementation”. Testing will consider two scenarios:

1. **Moderate throughput**: 3 clients, each throttled to (approximately) 1000 requests/second and generating (approximately) 10^4 requests.
2. **High throughput**: 10 clients at full speed, generating 10^5 to 10^7 requests in total. In this scenario your server needs to handle correctly more than 1000 requests per second, in average, generated by each client thread.

There will be at least 5 runs per scenario; the server is said to survive a run if it completes the run without crashing or producing a deadlock.
Your test harness will be assessed for functionality and that it produces results similar to what is presented in the report.

Marking Scheme

Grade	Server correctness and performance (60% weight)	Test harness and report (40% weight)	Marks
Fail	● The server does not compile or run. ● The server corrupts messages. ● The server crashes or deadlocks on every moderate throughput run.	● README is incomplete. ● The report has little or no description of experiments or no table. ● Test harness does not compile or run or produces results that differ wildly from the reported table.	0–39
Pass	● The server does not corrupt messages and survives at least one moderate throughput run.	● README complete. ● Report describes experiments in sufficient detail. ● Test harness produces data similar to the table in the report.	40–49
Fair Pass	● Server survives all but one moderate throughput runs.	● Average throughputs presented as graphs in a proper manner.	50–59
Good Pass	● Server survives all moderate throughput runs.	● Appropriate analysis and interpretation of the data.	60–69
Distinction	● Server survives all high throughput runs and serves more than 1000 requests per second per client thread.	● Plausible suggestions on how to change server design to improve performance.	70–84
Distinction+	● Average throughput of the server is equal or superior to the throughput of a standardised prototype. ● Every 1% throughput difference (rounded down) from the standardised prototype will result in one less mark from the maximum possible score.	● An investigation of which factors influence throughput, backed up by further experiments.	85+

To obtain a mark above distinction (Distinction+ in the table above) the student will need to submit evidence of the timing achieved with the standardised prototype and the timing achieved with his/her own version of the server and client. This can be achieved by capturing the screen output with the execution times for all the 10 executions required to successfully complete the section “Throughput experiments” and including them in the report. The code will be tested on a different computer, but the timing ratio between the reference prototype and the student’s server should roughly be similar. In case the ratio measured in this way is widely different, the student will be asked to demonstrate

the outcome of various runs on his/her computer in person, if appropriate, or in videoconference, sharing the computer's screen.

Submission Instructions

You must submit electronically to the module site on Blackboard by

Thursday, 8 December 2022, 3 pm

You will find the submission point in folder **Assessment**. You must submit a zip archive with the following content:

1. All source files of your server and test harness
2. The Visual Studio project files and the executable compiled; **DO NOT INCLUDE** temporary files or object files in your submission: these will only bloat the size of the submission, creating troubles with BlackBoard
3. Your throughput evaluation report as a PDF file, and a README file detailing:
    - The contents of your archive (one sentence per file suffices).
    - How to compile your code (if different from a Visual Studio Project).
    - How to run your server and test harness. This must include instructions on how to reproduce your throughput evaluation experiments.

DO NOT SUBMIT FILES RELATED TO THE COMPILATION PROCESS (temporary and object files). Submit only the files indicated above.
Your submission will be assessed by running and testing on a PC in room 9341. Therefore, the code must compile and run on these machines. Please make sure you test this before submitting.

In-Module retrieval

The module offers in-module retrieval for students who do not reach a passing grade. The specifications of the in-module retrieval will be the same as the original assessment. The date for the submission of the in-module retrieval will be in line with university guidance on in-module retrieval.

Fair practice

It is a requirement that the code you submit has been developed by you alone, without contributions from others. In order to ensure that submissions are unique, the automated marking system analyses the code submitted and reports on its similarity to other submissions. To ensure that your code is unique and has not been copied or developed in collusion with other students, the code will be automatically compared to the code submitted by all the other students on this module. If you develop your own code, you will naturally introduce differences that will ensure that your code is unique. Please ensure that you do not fall under suspicion of unfair practice by avoiding copying code. The consequences of plagiarising can be severe, including expulsion from your course and the University.