

CSE333 Exercise 11

Out: Friday, February 25

Due: Friday, March 4 by 11 am PST

Rating: 2 (note)

Goals

- Adapt server-side networking (TCP) in C/C++.

Problem Description

In the previous exercise, you created a client-side application to send over a local file. In this exercise, you will be implementing the corresponding server!

Write a C++ program that creates a TCP listening socket on a specified port number. Once a client connects, the program should read data from the client socket and write it to `stdout` until there is no more data to read from the client (*i.e.*, until EOF or a socket error occurs). Once all of the data has been read and written, the program should close the client socket and the listening socket, and then exit. Your program should accept the **port number** as the sole command-line argument.

An example execution of the completed application mirrors the use of `nc` from Exercise 10:

```
bash$ ./ex11 5555 > output.bytes
```

Provided Files

We have provided you with the following **four** source files, which can be downloaded from this web directory (`./ex11_files`) or with the commands:

```
bash$ wget https://courses.cs.washington.edu/courses/cse333/22wi/exercises/ex11_files/<filename>
```

- **SocketUtil.h** — Provides the public interface for various server-side networking utility functions. This is similar to the file from Exercise 10, but not the exact same. **DO NOT MODIFY.**
- **SocketUtil.cc** — Contains empty implementations of the utility functions declared in the header file.

- **ex11.cc** — Contains an empty `main` function for the server-side networking program.
- **Makefile** — Provided for your convenience in compiling the executable `ex11`.

Note: You will only submit `SocketUtil.cc` and `ex11.cc`.

Implementation Notes

Code Adaptation

Feel free to adapt sample code from lecture and section as part of your solution if it helps, but be sure you understand what your code does when you're done.

User Input

Since you will be reading in user input via command-line arguments, you should make sure your code handles various inputs from the user, which may be in an unexpected format.

Error Handling & Robustness

Remember that the networking methods can fail and should be handled appropriately in these cases. Make sure that you clean up system resources in *all* possible cases.

Testing Notes

Client & Server Setup

To test your program, you must run your server (`ex11`) first on the CSE Linux environment such as a specific `attu` machine:

```
bash$ ./ex11 <port> > output.bytes
```

Then, on some client machine, which does *not* need to be the same machine, send a file to your server using the `ex10` binary (either your own or the released sample solution). Assuming the server is running on `attu4` and listening on port 5555:

```
bash$ ./ex10 attu4.cs.washington.edu 5555 <filename>
```

Note that you can/should test on different types of files, such as text files (e.g., `ex11.cc`) and binary files (e.g., `ex10`).

Validation

Once the file has been transferred, you should confirm that the file was sent correctly. Instead of visually comparing the files in a text editor, which is nearly impossible for a binary file anyway, use existing utilities to make this easier:

- If your input file (the one passed to `ex10`) and output file (the one written by the redirect from `ex11`) are on the same machine (and possibly within the same directory), then you can use the `diff` utility to check for any discrepancies.

For example, if you sent the `ex10` binary, redirected the output to `output.bytes`, and they ended up in the same directory, then you can run:

```
bash$ diff ex10 output.bytes
```

There should be no output if the files match exactly.

- If your input and output files are on different machines, you can use the `md5sum` utility to produce a checksum digest for both and visually compare them.

For example, if you sent the `ex10` binary and redirected the output to `output.bytes`, then you would run the following on the server:

```
bash$ md5sum output.bytes
```

and the following on the client:

```
bash$ md5sum ex10
```

Style Focus

Don't forget that good practices from previous exercises still apply!

General

For the sake of our autograder, you may not modify `SocketUtil.h`, which also means that you should not modify the function signatures in `SocketUtil.cc`. However, feel free to replace the implementations of `WrappedRead()` and `WrappedWrite()` with yours from Exercise 10.

C/C++ Idioms

To work with the POSIX networking API, we, unfortunately, have to mix C and C++ idioms in our code. But you should still try to use C++ idioms whenever possible (e.g., use `cout` instead of `printf`, use C++ casting).


Submission

You will submit: `SocketUtil.cc` and `ex11.cc`.

Your code must:

- Compile without errors or warnings on CSE Linux machines (lab workstations, `attu`, or CSE home VM).

- Have no runtime errors, memory leaks, or memory errors (`g++` and `valgrind`).
- Be contained in the files listed above and compile with the provided Makefile.
- Have a comment at the top of your `.cc` files with your name(s) and CSE or UW email address(es).
- Be pretty: the formatting, modularization, variable and function names, commenting, and so on should be consistent with class style guidelines. Additionally, the linter shouldn't have any complaints about your code (`cpplint.py`).
- Be robust: your code should deal with hard-to-handle/edge cases and bogus user input (if there are any) gracefully.

Submit your code on  Gradescope (<https://www.gradescope.com>). Don't forget to add your partner if you have one.