Carleton University
Department of Systems and Computer Engineering
SYSC 3303 - Real-Time Concurrent Systems - Summer 2013

Project Specification

Teams will design and implement a file transfer system based on the TFTP specification (RFC 1350) that is available on the course website and that has been discussed in class. The system will consist of TFTP client(s) running on one or several computers, an error simulator, and a multithreaded TFTP server that runs on a different computer. The code will be written in Java, running one of the IDEs (e.g. JCreator or Eclipse) available in the SCE labs. *You must design your code to work in the lab environment provided!*

Note that there will be three separate programs: the client(s), the server, and the error simulator(s). Each program will run as a separate Win-32 process, and the programs will communicate via DatagramSocket objects. In “normal” mode, only the client and server programs will run. In “test” mode, all three programs will be used.

Your team's code should demonstrate good programming style, and be well documented. For examples of “industrial quality” Java code, have a look at Sun's Java coding conventions, which can be found on the Java resources Web site.

You must be able to run multiple main programs (projects) concurrently. See the Reference Material on the course web site for hints in doing this in Eclipse and JCreator.

Client Specification

The client will be implemented as a Java program that consists of one or more Java threads. The client will provide a *simple* user interface (*a GUI is neither required nor recommended*) that allows the user to input:

- the file transfer operation (read file from server, write file to server)
- the name of the file that is to be transferred

The client will then attempt to establish the appropriate connection with the server and transfer the file. After the current connection has been terminated (either because the file was transferred successfully or because an unrecoverable error occurred), the client should permit the user to initiate another file transfer. When the user indicates that no more files are to be transferred, the client should terminate.

The client should not support concurrent file transfers; for example, the client will not be able to concurrently transfer multiple files to and from one or more servers.

Server Specification

The server will be implemented as a Java program that consists of multiple Java threads. The server must be capable of supporting multiple concurrent read and write connections with different clients. To accomplish this, the server will have a multithreaded architecture. One thread will wait on port 69 for UDP datagrams containing WRQ and RRQ packets. This thread should:

1. verify that the received TFTP packet is a valid WRQ packet or RRQ packet
2. create another thread (call it the client connection thread), and pass it the TFTP packet; and

3. go back to waiting on port 69 for another request.

The newly created client connection will, as its name suggests, be responsible for communicating with the client to transfer a file.

Once started, the TFTP server will run until it receives a "shutdown" command from the server operator. Note that the server operator will type in this request in the server window. *It is neither desirable nor acceptable for a client to request that the server shutdown*. After being told to shut down, the server should finish all file transfers that are currently in progress, but refuse to create new connections with clients. The server should then terminate.

Error Simulation

Your error simulation code must be a completely separate program (or programs) from your client and server code. The error simulator communicates with the client and the server using DatagramSocket objects. The number of threads for this code is your choice. Note that it should be possible to run two or more clients as well as the server in error simulation mode.

User Interface

Both the client and the server should output detailed information about the current status of the system. While a real-world system would not give such detailed information, you may choose to have a less verbose option (e.g. "quiet" mode), but the verbose one *is* required (e.g. "verbose" mode). Keep the user interface (UI) simple. **A GUI is neither recommended nor required!** Do not develop a fancy UI comprised of instances of `JButton, JList, JComboBox, JCheckBox`, etc., etc., etc., for two reasons. First, developing a fancy UI is not the focus of this project. You should instead direct your efforts to ensuring that the programs correctly implement TFTP (how will you test ERROR packet creation and handling? How will you test the timeout/retransmit protocol?). Second, most of the methods in the Swing classes are not thread-safe. This means that mutual exclusion is not enforced if user-defined threads and Swing's event-dispatching thread invoke methods in Swing classes concurrently. You should be able to build an adequate UI using regular text windows or by using one or more `JTextArea` objects (possibly placed in `JScrollPane` objects) and various types of dialog boxes that can be created by class JOptionPane. The `append()` method in `JTextArea` is one of the few Swing methods that is guaranteed to be thread-safe. You should review the `JTextArea` API to see if there are any other thread-safe methods that may be of use. Some notes about JOptionPane will be posted on the course Web site. *If you choose to use any features not mentioned above, you must provide a detailed explanation in your documentation as to the research you did and the reasoning process you went through to convince yourselves that the additional features you used are indeed thread-safe*. **This will likely not be a trivial undertaking!**

Development Process

The project will be developed using an iterative, incremental process. The result of each iteration will be the release of an executable piece of software that constitutes a subset of the final TFTP client and server software. The software will be grown incrementally from iteration to iteration to become the final system. **Note that when submitting iteration "n", it is absolutely not acceptable to include code for iteration "n+1", etc.**

Iteration 0 – Establish Connections for File Transfer without Error Detection and

Correction

For this part, assume that no errors occur; i.e., no TFTP ERROR packets will be prepared, transmitted, received, or handled. Also, assume that no packets are duplicated in transit, and that no packets will be delayed or lost in transit, so the TFTP timeout/retransmit protocol is not supported.

In the lectures, we examined the timing diagrams for establishing a connection. Using these timing diagrams as a starting point, develop a skeleton TFTP client, error simulator, and server that run on the same computer. Your programs should permit clients to establish WRQ connections and RRQ connections with the server. For this iteration, do not implement the steady-state file transfer between the client and the server. It's sufficient to be able to demonstrate that clients can establish a RRQ connection and a WRQ connection, and that the server can establish multiple concurrent connections.

The error simulator will just pass on packets (client to server, and server to client), as we are not doing any error simulation at this point in time.

For each RRQ, the server should respond with DATA block 1 and 0 bytes of data (no file I/O). For each WRQ the server should respond with ACK block 0.

As noted earlier, the server must be multithreaded. For this iteration, each newly created client connection thread should terminate after it sends the appropriate acknowledgment to the client that requested the connection.

Also, there must be a nice way to shut down both your server and your client. CRTL-C is NOT a nice way!

Taking your solutions from assignment #1 and evolving that code into the first prototype of the TFTP system should be reasonably straightforward. The main difference for iteration 0 is that you will now need to create multiple threads in the error simulator and server code.

Work Products for Iteration #0:

- None - your first submission is Iteration #1 (below) which includes Iteration #0.

Iteration 1 – Implementation of File Transfer without Error Detection and Correction

The goal of this iteration is to extend the client, error simulator, and server programs to support steady-state file transfer. For this part, assume that no errors occur; i.e., no TFTP ERROR packets will be prepared, transmitted, received, or handled. Also, assume that no packets are duplicated in transit, and that no packets will be delayed or lost in transit, so the TFTP timeout/retransmit protocol is not supported.

In the lectures, we examined the timing diagrams for steady-state file transfer between a client and a server. Add steady-state file transfer capability to the client and server code developed in Iteration 0. As in Iteration 0, your server should create a new client connection thread for each connection with a client. You can have additional threads in the client and server, as long as you can justify them.

Again, the error simulator will just pass on packets (client to server, and server to client), as we are not doing any error simulation at this point in time.

Don't forget that there must be a nice way to shut down both your server and your client. CRTL-C

is NOT a nice way!

Ensure that you provide instructions as to how to run your code.

Work Products for Iteration #1:

- "README.txt" file explaining the names of your files, IDE used, set up instructions, etc.
- UML class diagram
- UCMs for a read file transfer and a write file transfer, including the error simulator
- Code (.java files, all required IDE files, etc.)

Iteration 2 – Adding TFTP Packet Format Errors (ERROR Packets 4, 5)

For this part, assume that errors can occur in the TFTP packets received, so TFTP ERROR packets dealing with this (Error Code 4, 5) must be prepared, transmitted, received, and handled. You may assume that no packets will be lost, delayed, or duplicated (see Iteration #3), and that there will be no File Input/Output errors (see Iteration #4).

Add support for ERROR packets as described above to the client and server code from Iteration #2. Note that you must parse every field of each TFTP packet for errors. You must submit code to enable us to see that your client and server deal with these errors – i.e. modify your error simulator. As many of these errors will happen very infrequently in practice, the best way is to build in an **extensive** test menu **in the error simulator** to test both the client and the server by forcing each error to occur, e.g. 0 : normal operation; 1 : invalid TFTP opcode on RRQ or WRQ; 2 : invalid mode, etc. We should be able to simulate any problem with any packet and any field within any packet. **Ensure that you provide instructions as to how to run and test your code.**

Note that there is not a one to one mapping between your error scenarios and the TFTP ERROR codes! For example, there are many ways to get ERROR code 4!

Work Products for Iteration #2:

- "README.txt" file explaining the names of your files, IDE used, set up instructions, etc.
- UML class diagram
- Timing diagrams showing the error scenarios for this and previous iterations
- Code (.java files, all required IDE files, etc.)

Iteration 3 – Adding Network Error Handling (Timeout/Retransmission)

Modify the client-server system from Iteration #3 to handle network errors. Packets can be lost, delayed, and duplicated. TFTP's "wait for acknowledgment/timeout/retransmit" protocol helps deal with this. Your program must contain the fix for the Sorcerer's Apprentice bug (i.e. duplicate ACKs must not be acknowledged, and only the side that is currently sending DATA packets is required to retransmit after a timeout). You may assume that there will be no File Input/Output errors (see Iteration #4).

You must submit code to enable us to see that your client and server deal properly with timeouts and retransmits – i.e. update your error simulator. As many of these errors will happen very infrequently in practice, the best way is to build in an **extensive** test menu **in the error simulator** to test both the client and the server that will allow you to test each situation, e.g. 0 : normal operation; 1 : lose a packet; 2 : delay a packet, 3 : duplicate a packet. Then you need to be able to select which packet to lose, delay or duplicate (e.g. RRQ, 2nd DATA, 3rd ACK, etc), and how much of a delay or space between duplicates. To repeat, you need to ensure that we can simulate the loss, delay or duplication of **any** packet, and the amount to delay / space the packets by (if applicable). **Ensure that you provide instructions as to how to run and test your code.**

Work Products for Iteration #3:

- "README.txt" file explaining the names of your files, IDE used, set up instructions, testing instructions, etc.
- UML class diagram
- Timing diagrams showing the error scenarios for previous iterations, and the timeout/retransmit scenarios for this iteration
- Code (.java files, all required IDE files, etc.)

Iteration 4 – Adding I/O Error Handling (ERROR Packets 1, 2, 3, 6)

For this part, assume that I/O errors can occur, so TFTP ERROR packets dealing with this (Error Code 1, 2, 3, 6) must be prepared, transmitted, received, or handled.

Add support for ERROR packets as described above to the client and server code from Iteration #1. Note that you must catch exceptions thrown by the your Java read/write code and translate the exceptions to the appropriate TFTP error codes. It should be possible to directly test all of these errors (e.g. file not found, etc.), i.e. you should not need to update the error simulator. **Ensure that you provide instructions as to how to run and test your code.**

Work Products for Iteration #4:

- "README.txt" file explaining the names of your files, IDE used, set up instructions, testing instructions, etc.
- UML class diagram
- Timing diagrams showing the error scenarios for this iteration
- Code (.java files, all required IDE files, etc.)

Iteration 5 – Implementation of File Transfer between Different Computers

Modify your client and server programs from Iteration #4 so that clients and the server can reside on different computers. The changes to Iteration #4 should be minimal. You just need to change the client's user interface to allow the user to specify the identity of the host where the server will run, and you'll need to modify the code that looks up the Internet address of the server's host. Think carefully about where the error simulator should run, and include this information in your documentation.

Final Work Products:

You must submit **hard and soft** copy of **all** of the following:

- Team number and team members
- Table of contents
- Detailed set-up instructions and testing instructions
- UCMs for read/write including the error simulator (from iteration #1)

- Timing diagrams showing all the error scenarios
- UML class diagram
- Code (.java files, all required IDE files, etc.)
- Please ensure that you provide adequate instructions for us to fully test your project and don't forget any files!
- Contact name and e-mail address in case we need to get in touch

Milestone Dates:

- Thursday, May 9th, 8pm: Team member names due (or you will be assigned a team)
- Monday, May 13th: Pre-Iteration 1 meeting
- Saturday, May 18th, 8pm: Iteration 1 due
- Wednesday, May 22nd: Pre-Iteration 2 meeting
- Saturday, May 25th, 8pm: Iteration 2 due
- Monday, May 27th: Pre-Iteration 3 meeting
- Saturday, June 1st, 8pm: Iteration 3 due
- Monday, June 3rd: Pre-Iteration 4 meeting
- Saturday, June 8th, 8pm: Iteration 4 due
- Monday, June 10th: Pre-Demonstration / Iteration 5 meeting
- Wednesday-Friday, June 12-14th: Project demo of Iteration 4 or 5 (your choice). Schedule of demonstrations will be determined and announced in class.
- Monday, June 17th, **3pm SHARP!!**: Deadline for submission of final work products (i.e. Iteration 5). Please bring your project to my office (ME4230).

Work products for each of iterations 1 to 4 are to be submitted using the project iteration submit program (different from the assignment submit program). Submit using your project team's number (one submission per team) by the date/time specified above. You can include any number of files in your submission. Ensure that all your files are accepted. Check the list of file types provided with the submit program information.

Project Marking Scheme:

Each of iterations #1-4 and the demo are marked out of 4. If a team gets 2/4 or higher on all iterations/demos, the mark for the final project is not adjusted. For **each** iteration and/or demo that gets 1/4, 1 mark is deducted from the final project mark. For **each** iteration and/or demo that gets 0/4, 2 marks are deducted from the final project mark.

The final deliverables are marked out of 20.

Weekly Meetings:

Each team will have a weekly 30min meeting with a TA to discuss their progress. The meetings are mandatory, and will be held during the labs. The meeting schedule will be posted on the web site.

- Each time a student **is late**, misses a meeting or does not contribute at a meeting, there will be a one mark deduction to their final project mark.
- In addition, if a team is not well prepared for a meeting, the team could have deductions from their final project mark.