Assignment 2: Socket Programming

COL334/672 : Computer Networks, Diwali'24

Deadline: 18 Sept 2024

Goal: The goal of this assignment is to familiarize you with socket programming, client-server communication, and scheduling algorithms. The assignment is divided into four parts, with the first two focusing on datagram sockets using C/C++ programming, and the remaining two focusing on scheduling algorithms. Appropriate hints are provided throughout the assignment. If you still have questions, you are encouraged to start a discussion on Piazza.

1 Word Counting Client

You need to implement client-server communication using TCP sockets. The goal of the client program is to read a list of words from a file located on the server and count the frequency of the words. Here is how the file reading works:

- The client establishes a TCP connection with the server, which is listening on a pre-defined IP and port. Once a connection is established, it remains persistent unless the client closes the connection.
- The client sends a request for the contents of the file with an offset indicating the location of the word in the file from which to start reading.
- The server has a memory-mapped file containing a comma-separated list of words. Once the server receives a request, it responds with k words starting from the offset mentioned in the request. However, instead of sending all the words in a single packet, the server sends p words in one packet.
- Once the entire file has been downloaded, the TCP connection is closed, and the client prints the frequency of the words with one word per line. The end of the file is denoted by a special word, `EOF`. For simplicity, assume this special keyword does not appear in the list of words.

API Details:

```
Client messages:
1. offset\n : Offset from which it needs words (Data Connection message). For example, '10\
    n' to read words starting from the 10th word. Here '\n' denotes the end of the packet,
    not any character in the payload.
2. Once the 'EOF' character has been read, the client should print the frequency of words,
    with each line corresponding to (word, frequency) ordered in dictionary order. For
    example, if the words were 'cat, bat, cat', the final output should be:
   bat, 1
   cat, 2
Server messages
1. $$\n : Offset greater than the number of words (Data Connection)
2. word_1\nword_2\nword_3\n...word_k\n : Normal Data Connection message. Here, 'k' is 10,
    and 'p' is 1. Ideally, 10 packets will be sent if there are 10 words available and
    fewer if there are fewer than 10 words.
```

1.1 Analysis

Run the experiment with different values of p, i.e. the number of words per packet, and log the completion time. Plot the completion time on y-axis for different values of p on the x-axis, ranging from 1 to 10, assuming k is 10. Please run the code 10 times for each value of p and plot the average completion time along with confidence intervals. Explain your observations. Additionally, discuss the factors that may influence the completion time, such as network bandwidth, latency, k, and p.

1.2 Submission

You should submit the code in a separate folder called `part1` containing `client.cpp`, `server.cpp`, and `Makefile`. In addition, assume a `config.json` file containing parameters, namely server IP, server port, k, p, and filename for the server to read words from. This config file is common for both client and server. It should also contain n, indicating the number of clients. We should be able to build and run the code using Makefile, i.e., `Make build`, `Make run` and `Make plot` (output a plot.png file). The server and client code should be written in C++, while the plotting code can be written in any language of your choice.

2 Concurrent Word Counting Clients

In this part, you need to extend your server to be able to handle multiple concurrent client connections. Note that the server is listening on the same port (You should appreciate the multiplexing allowed using TCP ports).

2.1 Analysis

Execute the word counter with different number of concurrent clients ranging from 1 to 32 (incremented by 4). Log the average completion time per client, then plot completion time per client against the number of clients. Analyze your observations: Does the completion time per client remain consistent or does it increase? Explain your observations.

2.2 Submission

You should submit the code in a separate folder called `part2` containing `client.cpp`, `server.cpp`, and `Makefile`. Like last time, assume a `config.json` file containing parameters. In addition, assume a parameter called `num_clients` indicating the number of concurrent clients. We should be able to build and run the code using Makefile, i.e., `Make build`, `Make run` and `Make plot` (output a plot.png file). The server and client code should be written in C++, while the plotting code can be written in any language of your choice.

3 Case of a Grumpy Server

Consider the case of 1 server and n clients. Assume p is 1 and k is 10. The server can have simultaneous active connections with multiple clients (like in the previous case) but prefers to serve only one request at a time. If the server receives a new request while serving an existing request, the server halts both requests and sends a HUH! message to both clients, indicating them to come after some time (think of it as a grumpy uncle!). The existing client is also expected to discard any communication it received during that specific request. The clients then have a distributed protocol to retry communication with server, with no direct communication among the clients. [Any resemblance to a multiple access medium is NOT coincidental!]

Concurrency condition: The server should send HUH! message to a pair of clients C1 and C2 when request arrival of C2 is within (t_1^{C1}, t_2^{C1}), with t_1^{C1} being C1's request arrival and t_2^{C1} being C1's request completion time.

Your task is to implement the following three distributed client-server communication protocols:

1. **Slotted Aloha Protocol**: Each client decides to send a request with probability *prob* at the start of the slot. Assume each slot is T ms long and begins at a UNIX timestamp (in milliseconds) that is a multiple of T ms. There is no clock synchronization across nodes. Also, assume p = $\frac{1}{n}$, which is known to each client.
2. **Binary Exponential Backoff (BEB)**: Once two clients receive a HUH! message from the server, they start a binary exponential backoff counter. The client waits for $i \times T$ ms where i is a random number between (0, 2^k-1) in the k^{th} attempt.
3. **Sensing and BEB**: In this case, a client first politely asks the server if it is busy, i.e. already serving a request, by sending a control message (see description below). If the server responds with $IDLE$, the client sends a data request to the server. Otherwise, it waits for $T\ ms$ before asking again. Note that such a request does not change the status of the server. The status is only changed if it receives a data request. If it receives a HUH! message during the transmission, the client reverts to BEB as described before.

```
\textbf{API Details}
Server:
HUH!\n: if it receives a new request while serving an existing request
IDLE\n: if it is not serving any client (only part 3)
BUSY\n: if it is serving a client (only part 3)

Client:
BUSY?\n: to check if the server is busy (only part 3)
```

3.1 Analysis

Run your program for different values of n across the three communication protocols and log the average completion time per client. Generate a single plot where x-axis represents the number of clients and y-axis represents that average completion time per client. Explain your observation across the three protocols. Also, briefly explain the impact of parameter changes in each protocol. For example, what happens when you change *prob* and slot length in slotted ALOHA?

3.2 Submission

You should submit the code in a separate folder called `part3`, which should contain the files `client.cpp`, `server.cpp`, and `Makefile`. In addition, assume a `config.json` file like previous parts. We should be able to run the code as follows:

- `make build`: builds the code
- `make run-aloha`, `make run-beb`, and `make run-cscd` runs communication protocol 1, 2, and 3, respectively, and outputs the average completion time per client in each case. `make run`: Runs all three procotols and logs the average completion time per client in as three comma-separated values.
- `make plot` outputs a `plot.png` file with the plot mentioned in Section 3.1.

Implementation hints

- To accurately detect concurrent requests (i.e., collision) at the server, you can use a shared struct variable that includes server's current status, the socket ID being served, the start time of the request being served, and the last time a concurrent request was detected. You can use these variables to detect concurrent request as follows:
  - If the server's current status is busy, or
  - If the last time a concurrent request was detected is greater than the current request arrival time. *Why might this happen?* Context switching. Consider three concurrent requests. Ideally all three should receive a HUH!. However, if second request is context switched right after its arrival is timestamped, a collision may be detected between the first and third requests, resulting in a HUH! being sent to them and the server state switching to idle. When the second request resumes, the server would appear idle and serve the second request. [Feel free to start a discussion on Piazza regarding this point.]
- Implementing `BUSY?` logic: You can again use the globally shared variable indicating the status of connection. The variable will be atomically changed by a client data request process when it begins and ends. You may use CAS for this purpose.

4 Case of Friendly Server

Again, consider the case of one server and n clients. Like in the previous scenario, the server can only serve one data request at a time, even though it can maintain multiple concurrent active connections. However, this time, the server schedules concurrent data requests using a centralized scheduling algorithm (a friendlier version!). You are required to implement the following scheduling policies:

1. **FIFO**: Implement *First In First Out*(FIFO) scheduling at the server. In this policy, data requests are served in the order of their arrival.
2. **Fair Scheduling**: FIFO scheduling may have fairness issues, as a single client could monopolize the server by sending multiple concurrent requests. To address this, implement a round-robin scheduling policy at the server.

4.1 Analysis

- Run your program for different values of n across the two scheduling policies. Plot a graph with n on the x-axis and average completion time on the y-axis for the two scheduling policies. Explain your observations. Also, compare the completion time with decentralized scheduling algorithms in Section 3.
- Emulate a scenario with 10 clients, where one rogue client sends 5 concurrent requests at a time while others send only one request. Log the average completion time per client as well as the `Jain's fairness index` for both scheduling policies. Explain your results.

4.2 Submission

You should submit the code in a separate folder called `part4`, which should contain the files `client.cpp`, `server.cpp`, and `Makefile`. In addition, assume a `config.json` file like previous parts. We should be able to run the code as follows:

- `make build`: builds the code

- `make run-fifo`, `make run-rr` runs the scheduling policy 1 and 2 respectively, and outputs the average completion time per client. `make run`: Runs both procotols and logs the average completion time per client in as two comma-separated values.
- `make plot` outputs a `plot.png` file with the plot mentioned in Section 4.1.
- `make fairness` runs the fairness study as described in Section 4.1 and reports the average completion time and `Jain's fairness index` as comma separated values in two lines with each line corresponding to one scheduling policy.

5 Submission Instructions

1. The assignment must be completed in pairs of two
2. You should only use pThreads for multi-threading.
3. Please adhere strictly to the provided API specifications for both the client and server. For autograding, your client files will be executed with the TA's server files, and vice versa.
4. Please conduct your observations on the provided test case file. You should attach all the plots and discussion in `report.pdf`.
5. Cheating will not be tolerated.
6. Should you have any questions or uncertainties, please post them on Piazza as a new post unless it is related to an existing post. Any email will not be entertained.
7. **Submission :** Submit a single zip file named *entrynum*1_*entrynum*2.zip, containing all the required files. Eg.,

```
2020CS50444_2020CS10000.zip
 part1
    server.cpp
    client.cpp
    config.json
    Makefile
    any auxilliary files
 part2
    ..
 part3
    ..
 part4
    ..
 report.pdf
```